UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Telecomunicació
i Aeroespacial de Castelldefels

European Space Agency
Agence spatiale européenne

TREBALL DE FI DE GRAU

TFG TITLE: AOCS of a nanosatellite type CubeSat 1U based on reaction wheels

DEGREE: Grau en Enginyeria d'Aeronavegació

AUTHOR: Victoria Skiba Maldonado

ADVISOR: Julio Gallegos Alvarado

SUPERVISOR: Pilar Gil Pons

DATE: July 9, 2020

Title: AOCS of a nanosatellite type CubeSat 1U based on reaction wheels
Author: Victoria Skiba Maldonado

Advisor: Julio Gallegos Alvarado

Supervisor: Pilar Gil Pons

Date: July 9, 2020

Overview

CONTEXT: The European Space Agency is currently developing a CubeSat, aimed to transmit to Earth well-defined reference signals which will be used to calibrate ground-based Cosmic microwave background (CMB) polarimeters. In order to establish reliable communication with the ground based stations, this CubeSat's orientation must be highly accurate, and the required accuracy implies the development of an Attitude Orbit Control System (AOCS).
AIM: The goal of this Bachelor's Thesis is to design and test an AOCS system in one rotation axis.
METHODOLOGY: The AOCS proposed is based on a reaction wheels system. The detailed design involves the definition of components, as well as the design of a proportional integral and derivative (PID) controller to direct the system and test its performance. Given the impossibility of working in ESA laboratories due to COVID-19 restrictions, the tests we actually developed were an adapted simplified version of the ones originally planned.
RESULTS: We have analysed the behaviour of a simplified version of the an AOCS in one axis. Over 100 tests were carried out for different parameters, in order to analyse the system components' performance, as well as the effects of the input parameters. All the elements except the launchpad showed a good performance. The results of our parametric analysis were hampered by friction (much higher than that expected in the real satellite), however, the information we obtained can be useful as a guideline for future laboratory experiments. Once the complete design and successful tests are performed, the system will be on board of the 3U ESA CubeSat, and will orientate and control the nanosatellite into the desired position.

Títol: AOCS of a nanosatellite type CubeSat 1U based on reaction wheels
Autor: Victoria Skiba Maldonado

Director: Julio Gallegos Alvarado

Supervisor: Pilar Gil Pons

Data: 9 de julio de 2020

Resum

CONTEXTO: La Agencia Espacial Europea está desarrollando actualmente un CubeSat destinado a transmitir a la Tierra señales de referencia bien definidas que se utilizarán para calibrar los polarímetros de fondo de microondas cósmicos (CMB) basados en tierra. Para establecer una comunicación fiable con las estaciones terrestres, la orientación del CubeSat debe ser muy precisa y para ello, es necesario disponer de un sistema de control de actitud y guiado (AOCS).
OBJETIVO: En este TFG se propuso desarrollar un sistema AOCS simplificado. que pueda funcionar en un eje de giro.
METODOLOGÍA: Para lograr la precisión requerida, se propuso el desarrollo de un sistema AOCS basado en ruedas de reacción. El diseño detallado implica la definición de componentes, así como el diseño de un controlador PID para dirigir el sistema y probar su rendimiento. Dadas las restricciones impuestas ante la crisis del COVID-19, los experimentos no se pudieron realizar en laboratorios, y por lo tanto se tuvieron que adaptar y simplificar respecto a la propuesta inicial.
RESULTADOS Hemos analizado el comportamiento de una versión simplificada del sistema AOCS en un eje. Hemos realizado alrededor de 100 experimentos con diferentes parámetros, para determinar el funcionamiento de los componentes y los efectos de los parámetros de entrada. A pesar de que nuestros resultados se han visto afectados por los efectos de la fricción, que se espera que será mucho menor en el CubeSat, los resultados obtenidos pueden servir de referencia para futuros experimentos realizados en el laboratorio y con el equipamiento adecuado. Una vez que se realice el diseño completo y las pruebas sean exitosas, el sistema estará a bordo del CubeSat 3U de la ESA. Cuando esté en funcionamiento, el sistema orientará y controlará el nano-satélite en la posición deseada.

I would like to express my gratitude to all the people that have helped me during my Bachelor's thesis. Professor Pilar Gil for her guidance and encouragement, also my supervisor Julio Gallegos for his advice and for helping anytime, even during the weekends, and finally Gonzalo Vescovi for his continuous support. Also I would like to thank Maria Garcia for trusting in me and letting me start my traineeship at ESAC.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. ACRONYMS

ADCS Attitude Determination control system

AOCS Attitude Orbit Control System

API Application Programming Interface

CMB Cosmic Microwave Background Radiation

CMGs Control Momentum Gyroscopes

COT Commercial of-the-shelf

DC Direct Current

EPS Electronic Power System

ESA European Space Agency

ESAC European Space Astronomy Centre

ESC Electronic Speed controller

IDE Integrated Development Environment

ISS International Space Station

LEO Low Earth Orbit

MCU Main Control Unit

MEO Medium Earth Orbit

MOSFET Metal-Oxid Semiconductor Field Effect Transistor

MTs Magnetorquers

NASA National Aeronautics and Space Administration

OBC On Board Computer

ODE Ordinary Differential Equation

PID Proportional Integrator and Derivative controller

P-POD Poly-PicoSatellite Orbital Deployer

PWM Pulse Width Modulation

RWs Reaction Wheels

CHAPTER 2. INTRODUCTION

In recent years the number of launched nanosatellites has dramatically increased (see figure 2.1), leading to a new era of inexpensive satellites and full accessibility to space exploration. Space missions, which were restricted to the largest space agencies, are now open to small companies and educational institutions (see figure 2.2) [1]. CubeSats are slowly allowing space democratization by offering the chance to access space for a very inexpensive price.

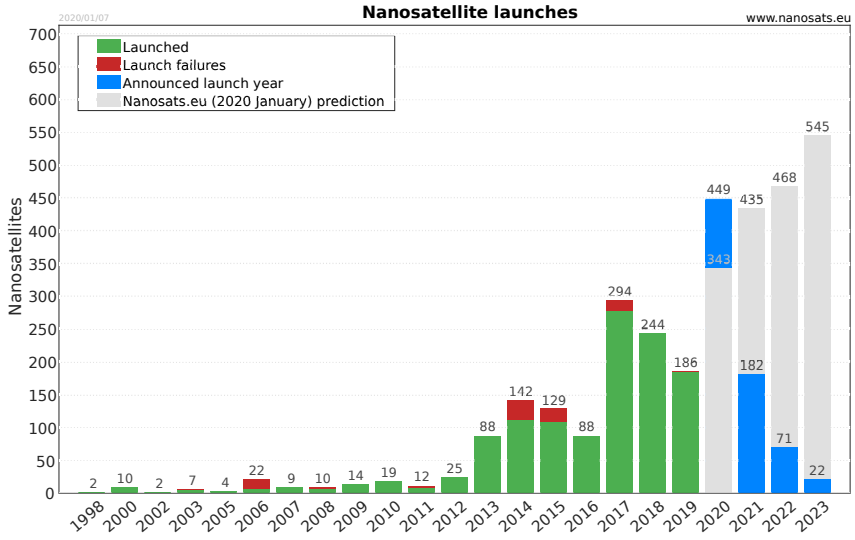


Figure 2.1: Nanosatellite launches over the years since the first launch. Credit: www.nanosats.eu.

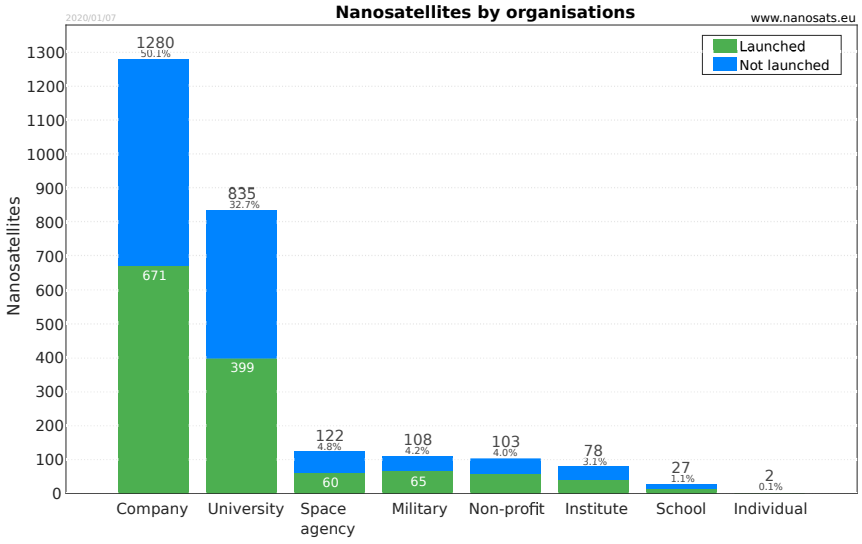


Figure 2.2: Different types of organisations launching Cubesats. Credit: www.nanosats.eu.

A satellite consists of several subsystems, depending on the goals of their specific mission. The power supply system, the attitude and orbit control system, and the telemetry and communications subsystem are crucial for the operation and performance of satellites. The attitude and orbit control system (AOCS) is one of the key subsystems for new mission concepts such as deep-space missions, rendez-vous and docking, formation flying... The European Space Agency has only launched one CubeSat with AOCS technology and reaction wheels in march 2020, hence the importance of the related research and the efforts to improve this technology.

2.1.. The European Space Agency

The work presented in this thesis is part of a CubeSat project run by the European Space Agency (ESA). With its 22 member states, ESA is Europe's intergovernmental space organisation for both manned and unmanned spaceflight and Earth observation. Headquartered in Paris, ESA has several research and test sites in Europe. Of these places, the largest are the European Space Research and Technology Centre in Noordwijk, Netherlands, where most of ESA's spacecraft are developed and thoroughly tested before launch, and the European Space Operations Centre in Darmstadt, Germany, which hosts the main mission control centre and coordinates development of ground systems. Notable missions by the European Space Agency include Huygens (the first space probe to land on another moon, Saturn's moon Titan), XMM-Newton (which investigates interstellar X-ray binaries, supermassive black holes and dark matter), Rosetta. (the first space probe to land on a comet), Gaia ("The Billion Star Surveyor"), Hubble Space Telescope and the soon to be launched James Webb Space Telescope (both of which are in collaboration with NASA). As well as deep space astronomy and planetary exploration, ESA is involved in manned spaceflight, namely scientific and supply missions to the International Space Station (ISS), and earth observation.

The CubeSat project, which provides the context of the present work, is being conducted at the European Space Astronomy Centre (ESAC) in Madrid. This site is the scientific centre of ESA and hosts the operation centres for planetary and astronomy missions as well as data centres.

2.2.. The concept of CubeSats

CubeSats (See figure 2.3) are a type of miniature cubic satellites, that if measure less than 10cmx10cmx10cm and weight less than 10kg per unit can be considered nanosatellites [1].

[1] Depending on the mass of small satellites they can be classified as follows:

- Minisatellite (100–500 kg)
- Microsatellite (10–100 kg)

Figure 2.3: ESA CubeSat model. CREDIT: ESA

CubeSats where born in 1999 when professor Jordi Puig-Suari proposed the first CUbeSat reference design, with the aim of enabling graduate students to design, build, test and operate in space a spacecraft with capabilities similar to that of the first spacecraft, Sputnik. Due to standarized shape and dimensions, they usually carry COTS (commercial of-the-shelf) components which allow to reduce significantly the prize of its production. CubeSats are frequently used to prove technologies for big satellites which do not justify the costs of one. They are also commonly used for space observation, educational tools, deploying small payloads and lately for *Internet of Things* purposes.

CubeSats usually have a short service life (around 1 year or less), as they are deployed in Low Earth Orbits (LEO) and Medium Earth Orbits (MEO). Due to relatively high atmosferic friction they end sprialing into the Earth surface and disintegrating during the reentry.

Launch costs are relatively low because of the limited weight per unit and because of the use of standardised deployment systems such as the Poly-PicoSatellite Orbital Deployer (P-POD) (See figure 2.4) . They can be launched either attached to a bigger satellite, from P-PODs along with other CubeSats, or from the ISS.

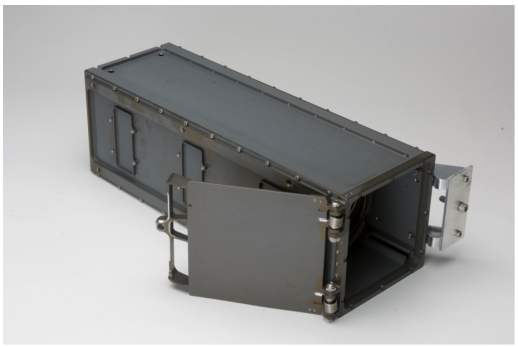


Figure 2.4: P-POD Poly Picosatellite Orbital Deployer. CREDIT: www.nasa.gov.

- Nanosatellite (1–10 kg)
- Picosatellite (0.1–1 kg)
- Femtosatellite (0.01–0.1 kg)

CubeSats can be configured as a single cubic unit (1U) of maximum $1.3\,\mathrm{kg}$ or as multiples of that on 2U, 3U, 6U or 12U platforms. Currently the number of launched 3U CubeSats represents the vast majority of all the launched nanosats (See figure 2.5).

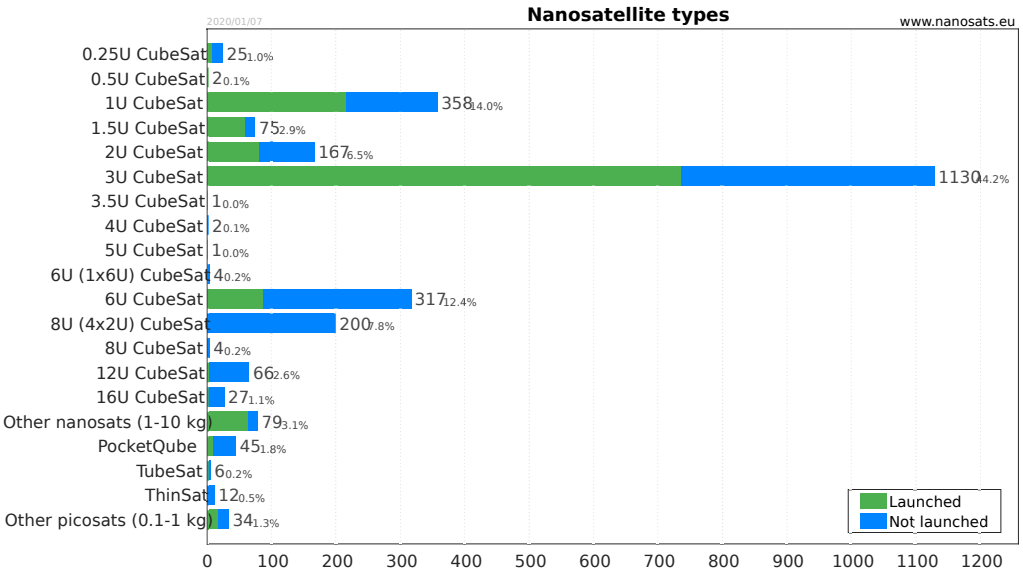


Figure 2.5: Types of launched CubeSats. CREDIT: www.nanosats.eu.

2.3.. The CubeSat Project at ESAC

The European Space Agency is currently developing a 3U CubeSat mission (yet unnamed) planned for 2021 at its Space Astronomy Centre near Madrid, Spain. The purpose of the proposed CubeSat is to send well-defined reference signals back to Earth for calibrating ground-based Cosmic Microwave Background Radiatiom (CMB) polarimeters. The CubeSat mission does not only include the spacecraft itself, but also a test bed for conducting system tests of the satellite's subsystems, especially the attitude determination and control system (ADCS), and a ground station for communicating with the satellite. Beforehand its launch, it is critical to thoroughly test the AOCS system that will be integrated in the 3U Cubesats.

The work to do at ESAC consists of the development of a 1U CubeSat prototype as a proof of concept regarding the subsystems of the 3U CubeSat and the test setup.

2.4.. Project goals and structure

The objective of this project is to define the elements that will conform the AOCS, and develop the PID analysis for the control of the CubeSat in the yaw axis.

Finally, we aim to test its performance.

The project is organized as follows:

- *Chapter 2: CubeSat Orientation* describes the main concepts necessary to understand spacecraft orientation.
- *Chapter 3: Spacecraft dynamics* presents the main equations that describe the motion of a spacecraft, as well as the main attitude control devices.
- *Chapter 4: Electronics* describes each component of the CubeSat's payload.
- *Chapter 5: Qualitative Test of the control in the yaw angle* presents a first test performed in order to try the test bed and the PID. Only qualitative results are considered. The objective of this test is to see the performance of all the components and if it is necessary to reconsider any of the selected options.
- *Chapter 6: Performance test of the control in the yaw* angle is devoted to present a second test in which the main errors of test 1 are corrected. The aim of this new test is to quantify the performance of the PID.
- *Chapter 7: Conclusions and Future Work* presents a summary of our main results and conclusions, and draw guidelines for future related work.

CHAPTER 3. CUBESAT ORIENTATION

3.1.. Frames of Reference

The vector magnitudes which are needed to describe spacecraft orientation must be given with respect to a certain frame of reference. Let us recall that a frame of reference is defined by a point in space (the origin), and a set of Cartesian right-handed XYZ axis F(O, XYZ). Different frames of reference are relevant in this context, and we will now describe the Body frame of reference and the Horizontal frame of reference.

3.1.1.. Body Frame of Reference (F_b)

- Origin at the centre of mass of the aircraft.
- X_B axis along the mean chord line of the aircraft (positive towards the head).
- Z_B axis perpendicular to X_B (positive pointing downwards). Note that X_B and Z_B are in the plane of symmetry of the aircraft.
- Y_B axis perpendicular to X_B and Z_B axes. The system follows the right hand's rule.

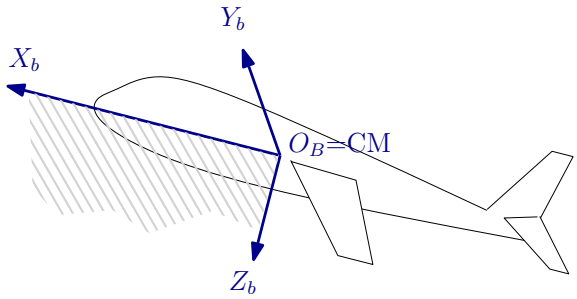


Figure 3.1: Description of the Body SR. Credit: notes of Classical Mechanics, EETAC.

Note that the axes X_B and Z_B define a plane of symmetry of the aircraft (see Figure 3.1).

3.1.2.. Horizontal Frame of Reference (F_H)

- Origin at the centre of mass of the aircraft.
- Z_H axis parallel to weight direction (and thus perpendicular to X_H and Y_H).
- X_H is defined by convenience (perpendicular to Z_H, and depending on the specific situation). If no particular choice is preferred, X_H goes along the South-North direction.

- Y_H axis is perpendicular to Z_H and X_H. The system follows the right hand's rule.

Note that the horizontal frame of reference is equivalent to an inertial frame ($F_h \sim F_I$) as long as the planar Earth hypothesis holds. In a more realistic approach one should consider Earth's curvature and rotation.

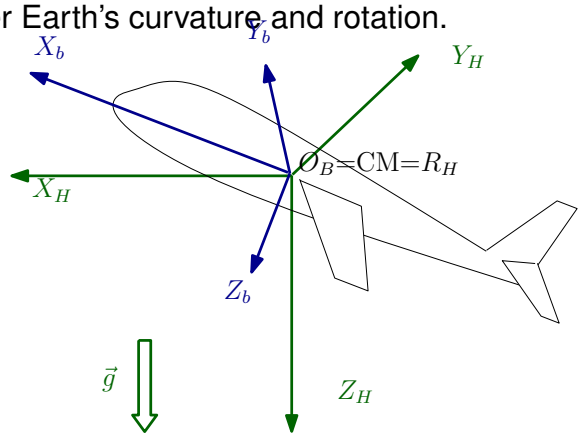


Figure 3.2: Relation between the Body SR and the Horizontal SR. Credit: notes of Classical Mechanics, EETAC.

Specifically, some magnitudes are naturally expressed in one frame of reference or another and, in any case, Newton's laws, necessary to describe the dynamics of flight, must be given in an inertial frame of reference. Therefore, it is necessary to have a method which will allow us to transform vector coordinates from one frame of reference to another. Assuming that the different frames have a common origin, such method merely involves rotation matrices as, after all, one can transform an orthogonal frame into another simply by performing a rotation (see Figure 3.2).

3.2.. Euler Angles

3.2.1.. Euler angles description

The Euler angles are three angles that, when provided about specific axes and in a specific order, are used to describe the orientation of an object (e.g an aircraft) with respect to a fixed reference frame.

The Euler angles in the Tait-Bryan notation are yaw(ψ), pitch(θ) and roll (ϕ) (see Figure 3.3).

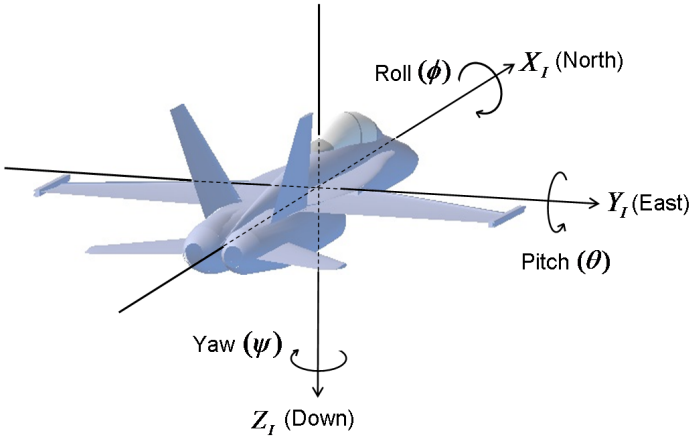


Figure 3.3: The inertial frame. CREDIT: Wikipedia

The rotations matrices associates to each Euler angle for yaw, pitch and roll are, respectively, the following:

$$R_\psi = \begin{pmatrix} cos\psi & sin\psi & 0 \\ -sin\psi & cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.1}$$

$$R_\theta = \begin{pmatrix} cos\theta & 0 & -sin\theta \\ 0 & 1 & 0 \\ sin\theta & 0 & cos\theta \end{pmatrix} \tag{3.2}$$

$$R_\phi = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos\phi & sin\phi \\ 0 & -sin\phi & cos\phi \end{pmatrix} \tag{3.3}$$

These 3 matrices multiplied in the indicated order give the complete orientation matrix that describes coordinate transformations from an inertial frame to the body frame:

$$R_\psi R_\theta R_\phi = \begin{pmatrix} cos\psi cos\theta & cos\theta sin\psi & -sin\theta \\ cos\psi sin\phi sin\theta - cos\phi sin\psi & cos\phi cos\psi + sin\phi sin\psi sin\theta & cos\theta sin\phi \\ sin\phi sin\psi + cos\phi cos\psi sin\theta & cos\phi sin\psi sin\theta - cos\psi sin\phi & cos\phi cos\theta \end{pmatrix} \tag{3.4}$$

3.2.2.. Gimbal Lock

A gyroscope is a spinning device which maintains its orientation in space. Since the gyroscope is nested within 3 gimbals (see Figure 3.4), it acts as 3 gyros. A slight rotation will cause the gyro to react in order to quickly recover its orientation. This action of the gyro can be recorded in order to measure the orientation of the spacecraft in space.

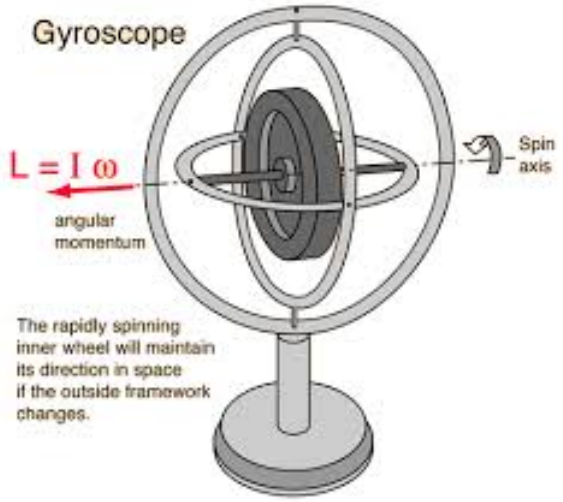


Figure 3.4: Gimbals Gyroscope.CREDIT: Wikipedia.

A disadvantage of using Euler angles is the so called Gimbal lock. This phenomenon occurs when the pitch is $\pm 90^{o}$ (See Figure 3.5). Once the yaw and roll angles align, both stay blocked. Under this circumstance only the pitch is able to move, and the spacecraft has no longer self-control, as the pitch gimbal continues moving by its own inertia.

The gimbal lock is a very dangerous situation. In the Apollo 13 mission, it caused a decontrolled spinning of the spacecraft. In the 1970's the only way to prevent gimbal lock was to avoid reaching a $\pm 90^{o}$ pitch situation by warning the pilots and acting manually.

A more efficient way of avoiding gimbal lock is the use of Quaternions, described in the following section.

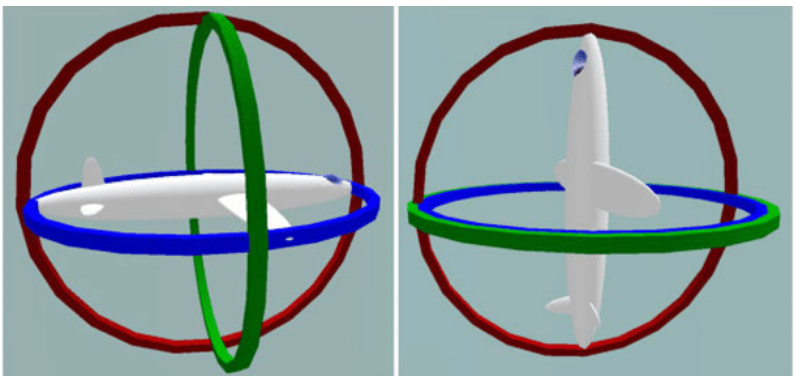


Figure 3.5: Left: Euler angles in normal conditions. Right: Euler angles when the pitch angle is close to $\pi/2$ and both the yaw and roll gimbals aligned (gimbal lock). CREDIT: Wikipedia.

3.3.. Quaternions

The use of quaternions is an alternative approach to Euler angles for calculating 3D rotations. Quaternions are used throughout this work to describe rotations in a defined reference frame, as well as a satellite's orientation and transformations into different frames. Just as vectors can be expressed as number triplets, quaternions are represented by 4-dimensional numbers as follows:

$$q = q_0 + q_1\vec{i} + q_2\vec{j} + q_3\vec{k} \tag{3.5}$$

where a is the scalar part, $bi + cj + dk$ is the vector part, and $\vec{i}$, $\vec{j}$ and $\vec{k}$ are the unitary vectors associated to an orthonormal basis in 3D, which must comply with the following expressions:

$$\vec{i}^2 = \vec{j}^2 = \vec{k}^2 \tag{3.6}$$

$$\vec{i} \cdot \vec{j} \cdot \vec{k} = 1 \tag{3.7}$$

Alternatively, quaternions may be expressed simply as a set of 4 numbers:

$$q = [q_0, q_1, q_2, q_3] \tag{3.8}$$

The most basic operations with quaternions may be summarized as follows:

- Sum: $p + q = [p_0 + q_0, p_1 + q_1, p_2 + q_2, p_3 + q_3]$.
- Scaling (with real number a): $a\Delta q = [aq_0, aq_1, aq_2, aq_3]$.
- Norm: $l = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}$ ($l = 1$ for unit quaternions).
- Conjugation: $q^* = [q_0, -q_1, -q_2, -q_3]$.
- Inverse: $q^{-1} = \frac{q^*}{l} = \frac{[q_0, -q_1, -q_2, -q_3]}{\sqrt{q_0^2+q_1^2+q_2^2+q_3^2}}$. ($q^{-1} = q^*$ for unit quaternions).
- Product:

$$p \otimes q = \begin{bmatrix} p_0q_0 & -p_1q_1 & -p_2q_2 & -p_3q_3 \\ p_0q_1 & +p_1q_0 & +p_2q_3 & -p_3q_2 \\ p_2q_0 & -p_1q_3 & +p_2q_0 & +p_3q_1 \\ p_3q_0 & +p_1q_2 & -p_2q_1 & +p_3q_0 \end{bmatrix}.$$

- Conjugate of product: $(p \otimes q)^* = q^* \otimes p^*$.

3D rotations can be described as unit quaternions: If $R(\theta, \omega)$ is the rotation in 3D around an axis described by the vector $\omega = (\omega_x, \omega_y, \omega_z)$, with norm $\omega = \sqrt{\omega_x^2 + \omega_y^2 + \omega_z^2}$, and of angle θ, then $R(\theta, \omega)$ is described by the unit quaternion:

$$q_R = [\cos\frac{\theta}{2}, \frac{\omega_x}{\omega}\sin\frac{\theta}{2}, \frac{\omega_y}{\omega}\sin\frac{\theta}{2}, \frac{\omega_z}{\omega}\sin\frac{\theta}{2}]$$

Given the properties of rotation matrix algebra and quaternions, the inverse of a rotation $R^{-1}(\theta, \omega) = R(-\theta, \omega)$ and, as we saw above, $q^{-1} = q^*$.

Vectors in 3D can be described as quaternions: Given an arbitrary vector $\vec{v} = (v_x, v_y, v_z)$, the associated quaternion q_v is:

$$q_v = [0, v_x, v_y, v_z]$$

Quaternions (e.g. interpreted as rotations) can be applied on vectors:

$$[0, R(v)] = q \otimes [0, \vec{v}] \otimes q^*$$

Physically, the former equation represents a rotation or transformation of the vector coordinates from one frame 1 of reference to another frame 2, rotated a certain angle θ with respect to 1. According to Euler's rotation theorem, any sequence of arbitrary rotations about a fixed point can be expressed by a single rotation about the same point. Such rotation would be given by a quaternion.

Quaternions are used throughout this work to describe rotations in a defined reference frame as well as a satellite's orientation and transformations into different frames.
In addition to avoiding gimbal lock, an advantage of using quaternions instead of Euler angles is that, although being less intuitive, they are easier and more efficient to calculate by a computer. When using Euler angles one needs to rotate 3 times, which implies a product of 3 matrices, whereas the same rotation can be easily expressed by a single quaternion.

CHAPTER 4. FUNDAMENTALS OF SPACECRAFT ATTITUDE DYNAMICS

4.1.. Spacecraft Dynamics

For the analysis of spacecraft dynamics we start from Newton's second law of rotation, that relates the torque which is applied to a body to the time derivative of the angular momentum:

$$\vec{\tau}_I = \dot{\vec{h}}_I = \frac{d}{dt}(\mathcal{J}_I \vec{\omega}) \tag{4.1}$$

where $\vec{\tau}$ is the applied torque, $\vec{h}_I$ is the angular momentum, $\mathcal{J}_I$ is the inertia tensor of the body and $\vec{\omega}$ is the angular velocity vector. Note that all quantities are referred to an inertial reference frame of reference, thus subscripted by I. For the purpose of determining the dynamics of a spacecraft, it is useful to choose a body-fixed reference frame fixed to the spacecraft in our case. We define the body's center of gravity as the origin of this non-inertial reference frame. This choice of origin also yields the advantage that the rotational dynamics are decoupled from the translational dynamics, which means we only have to consider rotations about the centre of mass, without taking into account the translational motion of the spacecraft.

Furthermore, we define the body-fixed reference frame as a dextral set of principal axes of inertia. In this new reference frame, the inertia tensor becomes diagonal and is renamed as $\mathcal{I}$ to emphasise the principal axes:

$$\mathcal{I} = \begin{pmatrix} I_1 & 0 & 0 \\ 0 & I_2 & 0 \\ 0 & 0 & I_3 \end{pmatrix} \tag{4.2}$$

The angular momentum can now be expressed in the principal axes frame as:

$$\vec{h}' = I_1\omega_1\vec{e}_1 + I_2\omega_2\vec{e}_2 + I_3\omega_3\vec{e}_3 \tag{4.3}$$

where $\vec{e}_1$, $\vec{e}_2$ and $\vec{e}_3$ are the orthonormal basis of the body frame of reference. When the right hand side of Equation 4.1 is expressed in terms of variables given in the body frame of reference, it is rewritten as:

$$\vec{\tau} = \dot{\vec{h}'} + \vec{\omega} \times \vec{h}' = \frac{d}{dt}(\mathcal{I}\vec{\omega}) + \vec{\omega} \times \mathcal{I}\vec{\omega} \tag{4.4}$$

Let us assume now that the body considered here is a rigid body that experiences no deformation, regardless the existence of external torques and forces. In this case the components of $\mathcal{I}$ will be constant. Accordingly, the time derivative of the angular momentum can be written as:

$$\dot{\vec{h}} = \frac{d}{dt}(\mathcal{I}\vec{\omega}) = \mathcal{I}\dot{\vec{\omega}} \tag{4.5}$$

With this time derivative, equation 4.4 can be rearranged to obtain the rate of change of the angular velocity vector:

$$\dot{\vec{\omega}} = I^{-1}(\vec{\tau} - \vec{\omega} \times \mathcal{I}\vec{\omega}) \tag{4.6}$$

Note that the equation 3.4 is the general form of Euler's rotation equations. When given in the (body) principal axes frame, in which the moments of inertia are constant, these equations can be expressed as a set of three scalar first-order ODEs:

$$\begin{aligned} \tau_1 &= I_1\dot{\omega_1} + (I_3 - I_2)\omega_2\omega_3 \\ \tau_2 &= I_2\dot{\omega_2} + (I_1 - I_3)\omega_3\omega_1 \\ \tau_3 &= I_3\dot{\omega_3} + (I_2 - I_1)\omega_1\omega_2 \end{aligned} \tag{4.7}$$

In practice, the torques we implement to obtain specific angular velocity temporal rates in each axes will follow thew expressions above.

4.2.. Attitude control devices

In many space missions the payload, communication antennas or solar arrays require a stable target attitude of the spacecraft and the possibility of re-orientation manoeuvres (slewing) for different operational modes of the spacecraft. Attitude stabilisation can be achieved by passive actuation like gravity-gradient stabilisation or magnetic stabilisation. However, the pointing accuracy is limited and attitude manoeuvres are not possible with purely passive attitude control. Active attitude control allows a high pointing accuracy for these manoeuvres.

There are four commonly used types of active attitude actuators: thrusters, control momentum gyroscopes (CMGs), reaction wheels (RWs) and magnetorquers (MTs). Thrusters are part of a reaction control system. They are like small rocket engines that provide a wide range of available thrust levels and are mostly used in large spacecraft. Control momentum gyroscopes are a combination of a spinning flywheel and controllable gimbals that enable steering the angular momentum. The use of CMGs can lead to singularities, which means that for certain relative orientations there can be a loss of rotational degree of freedom.
In the following section it will be explained the attitude actuator used in this project, the reaction wheels.

4.2.1.. Reaction Wheels

Reaction wheels are attitude actuator units that consist of a flywheel that is mounted on an electric motor. They can affect the spacecraft's attitude by altering their rotational speed and thus changing their angular momentum. By conservation of angular momentum of the total system, this operation will transfer an angular momentum of the same size but with an opposite sign to the spacecraft. The biggest asset of RWs is their high pointing accuracy compared to other attitude

control strategies. Typical pointing accuracies between $\pm 0.001^{\circ}$ and $\pm 1^{\circ}$ can be achieved by reaction wheels. Other advantages are the independence of external influences (like magnetic field which magnetorquers require) and the fact that they do not use any fuel like thrusters do. However, they can saturate due to their limited speed range. As a consequence, a saturated reaction wheel can only create a torque in one instead of two anti-parallel directions. Additional effort is necessary to avoid saturation or to de-saturate the wheel.
A reaction wheel has an angular momentum that depends on its inertia I_{RW} and its rotational speed ω_{RW}:

$$\vec{h}_{RW} = \mathcal{I}_{RW}\vec{\omega}_{RW} \tag{4.8}$$

When using reaction wheels, this angular momentum has to be considered in the total angular momentum h in Euler's rotation equation. This yields an extended form of Equation 4.4, where h_{SC} denotes the spacecraft's angular momentum without the reaction wheels:

$$\vec{\tau} = (\dot{\vec{h}}_{SC} + \dot{\vec{h}}_{RW}) + \vec{\omega} \times (\vec{h}_{SC} + \vec{h}_{RW}) \tag{4.9}$$

where h_{RW} is the sum of the reaction wheels' momenta. Again, we assume that the inertia of the spacecraft I_{SC} is constant. Then we have:

$$\vec{\tau} = \mathcal{I}_{SC}\dot{\vec{\omega}} + \dot{\vec{h}}_{RW} + \vec{\omega} \times \mathcal{I}_{SC}\vec{\omega} + \vec{\omega} \times \vec{h}_{RW} \tag{4.10}$$

We also assume that the reaction wheel inertia $\mathcal{I}_{RW}$ is constant. Now we define the control torque applied by the reaction wheels as minus the time derivative of the angular momentum: $\vec{\tau}_c = -\dot{\vec{h}}_{RW} = -\mathcal{I}_{RW}\dot{\vec{\omega}}_{RW}$. Using this notation and rearranging equation 4.10 yields:

$$\dot{\vec{\omega}} = \mathcal{I}_{SC}^{-1}(\vec{\tau} + \vec{\tau}_c - \vec{\omega} \times \mathcal{I}_{SC}\vec{\omega} - \vec{\omega} \times \vec{h}_{RW}) \tag{4.11}$$

Note that the total angular momentum in a closed system in the absence of external torques remains constant. This applies to the angular momentum $\vec{h} = \vec{h}_{SC} + \vec{h}_{RW}$ of the closed system consisting of the spacecraft and reaction wheels, when no external torques are present. Therefore, a reaction wheel can not change the total angular momentum of the spacecraft, but it can exchange angular momentum with the rest of the system. For that reason, reaction wheels are sometimes referred to as momentum exchange devices, to distinguish their functional principle from other attitude actuators.
Reaction wheels can be operated as actual reaction wheels or as so-called momentum wheels. The former indicates that they have no bias rotational speed and are operated around the set value zero. As opposed to this, momentum wheels are operated around a nonzero set value and thus have a bias momentum.

CHAPTER 5. ELECTRONICS

This section describes all the components of the payload of the 1U CubeSat which is the basis of the present work. In this context, the payload of a satellite is the equipment specifically aimed to fulfill its mission. In this case, it is the AOCS system. In figure 5.1 it can be seen the internal components of the AOCS system mounted in a 1U CubeSat.

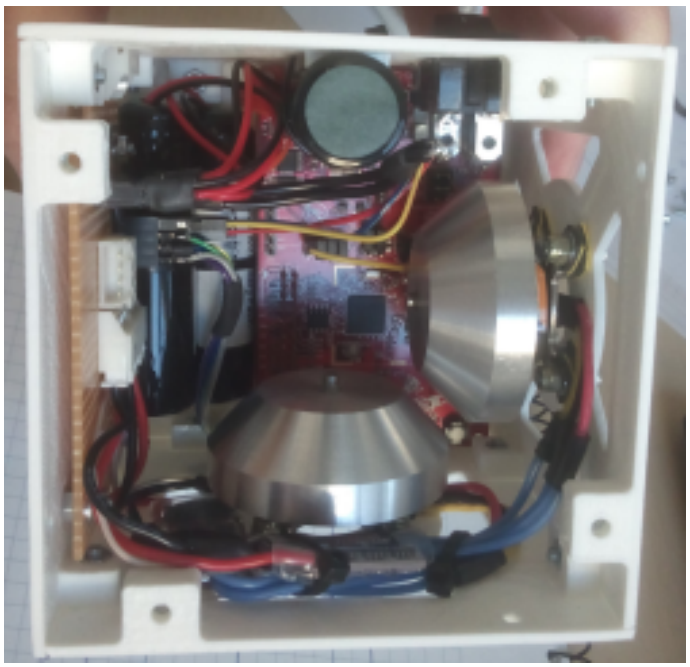


Figure 5.1: Internal components and subsystems.CREDIT: ESAC

5.1.. Electronic Components of the AOCS system

5.1.1.. On Board Computer

In order to control the attitude of the nanosatellite it is necessary to implement a proportional integral derivative (PID) to correct its position. The Main Control Unit (MCU) will be used for the acquisition of sensor data and to perform the necessary calculations for the PID controller.

The MCU chosen in the corrent work is the SimpleLink Wi-Fi CC3200-LAUNCHXL Launchpad [2]. Our choice was made for the following reasons:

- Small size. Fits easily into a 10cmx10cmx10cm nanosatellite.
- Large amount of Input/Output pins.
- Wide variety of embedded sensors.
- Programmable in C/C++.

- Fast 80MHz MCU.

Although the main reason to choose it is the fact that is has flight heritage, this means it has already flown in real space missions and proved to have a good performance.

The launchpad will be programmed in C/C++ with the open-source IDE *energia*. This Launchpad is also compatible with the I^2C protocol that is used to receive data from the gyroscope sensor.

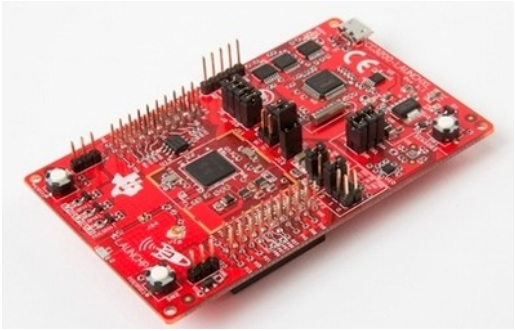


Figure 5.2: CC3200 Launchpad Texas Instruments.

5.1.2.. Reaction wheels

A reaction wheel is a device which is mounted onto an electrical brushless moto so that, when it rotates, it creates torque which allows to control the attitude of the device in which it is embedded. The torque on each axis allows to control the position by adjusting the power of the motor attached to the reaction wheel. Note that, because the reaction wheel and the motor are attached, both will move at the same velocity.

Once the CubeSat is in orbit the only external force that will actuate will be gravity. As gravity is a central force, its associated torque will be zero and thus its angular momentum will be conserved [1]. Therefore, if a wheel spins clockwise, the CubeSat will move counter-clockwise in response

[1] The fundamental equation of rotation estates the relation between external torques ($\vec{\tau}$) and angular momentum ($\vec{L}$) temporal variation to be $\frac{d\vec{L}}{dt} = \vec{t}au$

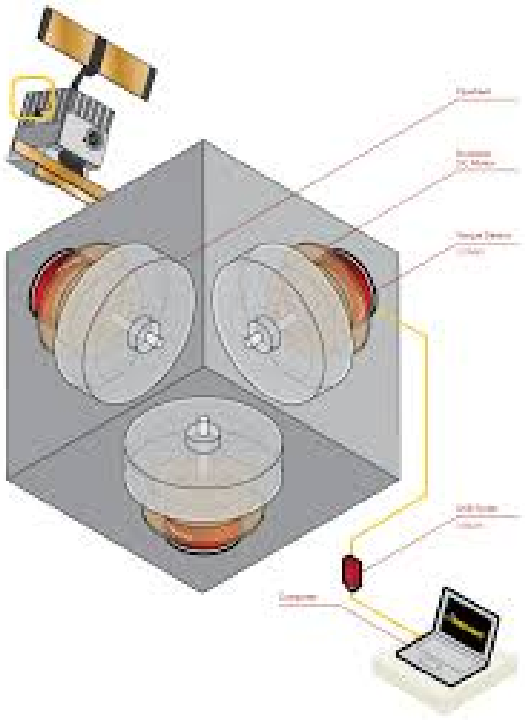


Figure 5.3: Structure of a CubeSat with reaction wheels.

Each reaction wheel (and its associated motor) will be mounted on a different face of the CubeSat to control each different axis. In order to allow any arbitrary rotation in 3D, 3 motors (in different axes of the CubeSat) are required. Figure 5.3 shows a sketch of the CubeSat, and how the reaction wheels are attached both to the CubeSat and to the motors.

The used reaction wheels are metallic, weight 51 grams each and their diameter is 5cm. They consist of a disk and a ring and thus, the total moment of inertia of a reaction wheel about each axis I_{T,X_i} is the sum of the moment of inertia of the disk, I_{disk,X_i} plus the moment of the ring, I_{ring,X_i}:

$$I_{T,X_i} = I_{disk,X_i} + I_{ring,X_i} \tag{5.1}$$

5.1.3.. Brushless DC motors

The chosen motor model is the *HobbyKing Donkey ST2204*[3] (see Figure 5.4). This model is light, small and although it is complex to control, this problem may be easily dealt with with the use of an external electronic speed controller (ESC). The ESC receives a digital signal and converts it into a triphasic one in order to feed the brushless motor. A disadvantage of this motor is the high friction it generates. This will lead to a less precise AOCS that will have to be corrected by the PID.

Figure 5.4: Brushless DC motor.

5.1.4.. Electronic speed controllers

The chosen speed controller model is the *HobbyKing XC-10A* (see Figure 5.5) [4].This one was chosen because it has flight heritage as well as the OBC. Each speed controller must be connected to a DC motor. The ESC allows to control the speed of the motor by PWM (Pulse width modulation) signals, which are square pulses in between 1 and 2 milliseconds with a 50Hz frequency. *HobbyKing XC-10A* has a reverse mode, although in this mode the power can't be controlled and the output is always constant. Even though the motors work with PWM pulses, they are not connected to the PWM pins of the launchpad. Instead, a specific library is used to control them.

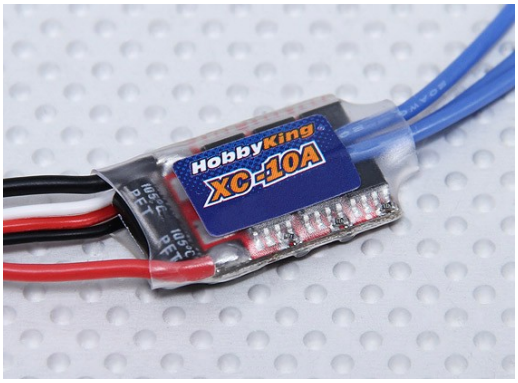


Figure 5.5: Speed controller.

5.1.5.. Gyroscope

In order to know the exact orientation of the CubeSat, it is necessary to have a gyroscope integrated The chosen gyroscope is the *BNO 055* from *Adafruit* [5] (see Figure 5.6). It returns the Euler angles which are used for the aforementioned purpose. This device also has integrated an accelerometer (which also provides information in 3 dimensions) and magnetometer (which gives information in 3 additional dimensions). The magnetometer is important because the information it provides can help correct small errors which build up in the acceleration, and may

cause drifts in absolute directions. As a consequence, our device can be considered a 9-axis inertial-motor sensor. It also includes a temperature sensor. The detailed outputs allow us to get or derive the following information:

- Absolute Orientation. Euler vector. Three axis orientation data based on a 360° sphere.
- Absolute Orientation. Four point quaternion output for more accurate data manipulation.
- Angular Velocity Vector. Three axis of 'rotation speed' in rad/s.
- Acceleration Vector. Three axis of acceleration (gravity + linear motion) in m/s^2.
- Magnetic Field Strength Vector.
- Linear Acceleration Vector. Three axis of linear acceleration data (after subtracting gravity) in m/s^2.
- Gravity Vector. Three axis of gravitational acceleration (after subtracting other sources of acceleration)in m/s^2.
- Temperature. Ambient temperature in degrees Celsius.

Figure 5.6: 9-axis gyroscope from Adafruit.

5.1.6.. Lithium-ion batteries

The energy required to allow the CubeSat operation is provided by 3 rechargeable batteries, one for each motor, of 3,7 V each. Technically, it is possible to use a single large battery instead, but in practice, it is easier to fit 3 batteries due to better positioning options, hence a better distribution of the mass center.

The 3U CubeSat that will be launched will be equipped with solar panels, therefore the batteries can be charged while in orbit. The cells, with a nominal voltage of 3.7 V each, are connected in series. This configuration yields a total nominal voltage of 11.1 V, defined as the main DC bus voltage.

5.1.7.. Electronic Power System (EPS)

a) **Source of Power to the Launchpad:**

While developing the AOCS system at Earth, the Launchpad obtains the power entry from a computer, as it is plugged through a USB port. While in Orbit the On Board Computer of the AOCS system needs a constant source of power. This will be obtained by means of the Electronic Power System (EPS), which will use the energy of the Li-ion batteries described above.

The batteries have an output of 3,7 V, while the Launchpad has to be feed with 3,3 V. Therefore it is necessary to convert this 3,7 V into the demanded voltage, while assuring that potentially harmful tension peaks will be avoided. For this purpose a linear voltage regulator filtered by a $22\,\mu F$ capacitor is used.

b) **Measurement of the batteries' voltage:**

it is crucial for the mission to know the amount of energy that the battery has left. The AOCS system is capable of knowing the voltage that the set of batteries have, as the Launchpad is in charge of monitoring the main DC Bus voltage through one of its pins. The voltage of the main DC bus is 11,1 V, so a voltage divider is going to adapt the voltage to the input pin. Furthermore the resistors used are $20\,k\Omega$ and $165\,k\Omega$, so the maximum drawn current will be very low. This will help to minimize the power dissipation thought the pin as it reads the voltage of the batteries. Any connection made to the Launchpad leads to power dissipation, thereby a MOSFET transistor will we used to cut the current flow whenever the Launchpad is not monitoring the battery. The transistor disconnects the voltage divider in between measurements.

Moreover two capacitors will be used to provide fast energy supply to the Launchpad when it is measuring the voltage of the batteries, as the objective is to reduce the current circulation. The quicker the measurement is, the lower the power dissipation.

5.2.. Electronic schematic and assembly

Figure 5.7 shows the schematic with the connections of the main components described above:

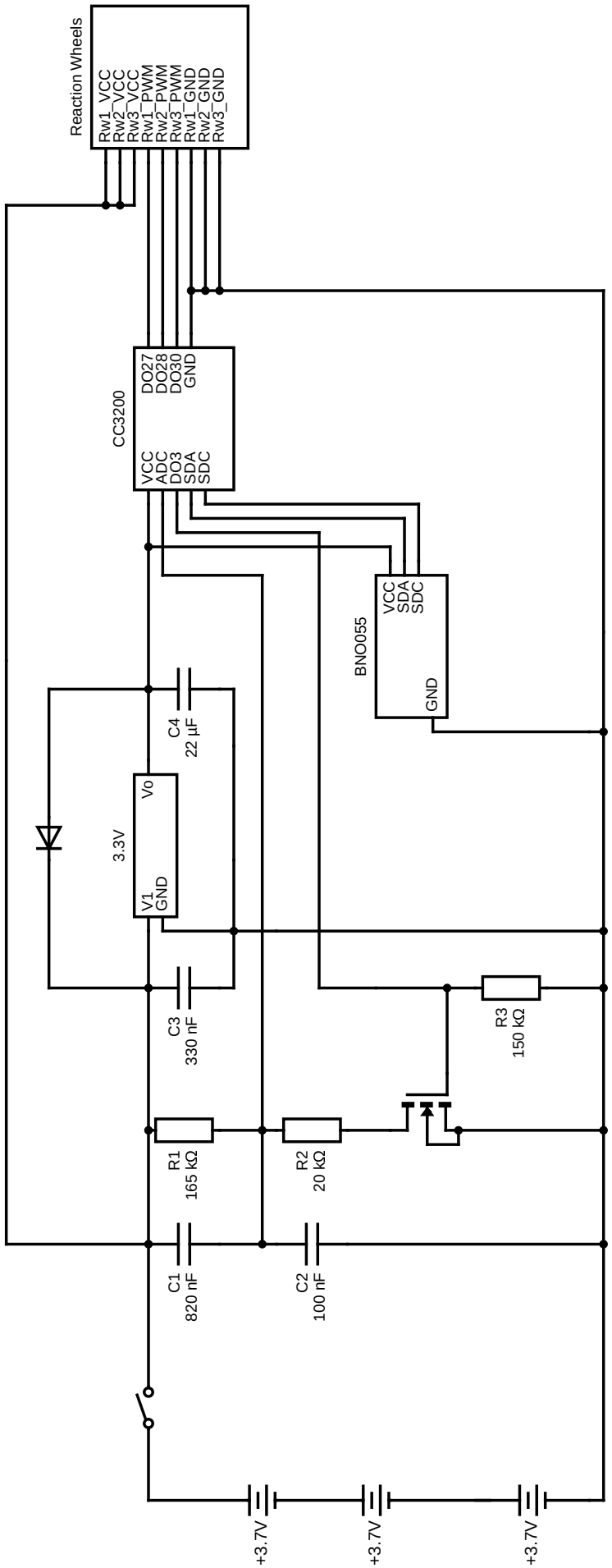


Figure 5.7: Electronic schematic showing the Motors, BNO 055 Sensor, ESC, Capacitors, Diodes, Resistances and batteries, following the standard notation.

Figure 5.8 shows the final assembly of the board that contains the BNO 055 sensor and the EPS. This board will be connected to the launchpad, batteries and motors.

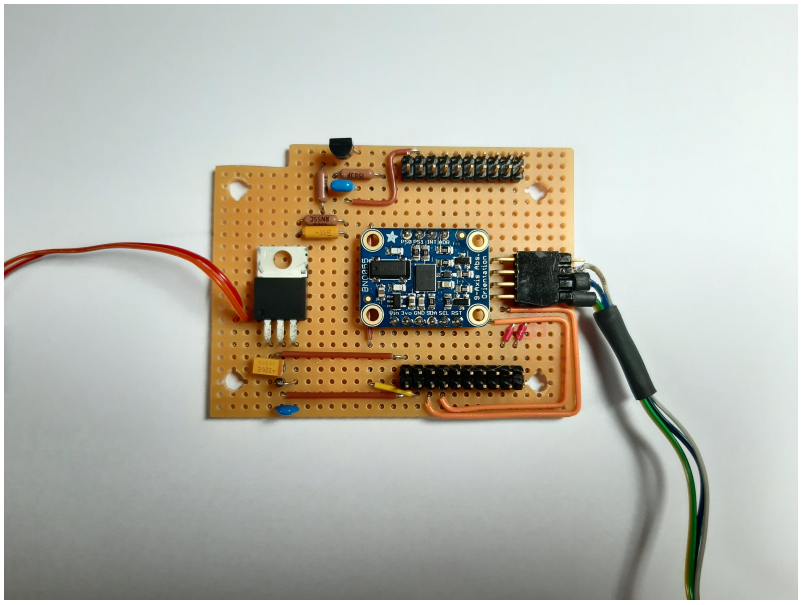


Figure 5.8: BNO 055 Sensor Board

CHAPTER 6. QUALITATIVE TEST OF THE CONTROL IN THE YAW ANGLE

In this section we describe the development of the first test of the electronics and firmware system.

Once the electronics has been assembled as described in Chapter 5, and the launchpad has been programmed in the IDE Energia (so that the motor and thus the reaction wheel can be controlled), it is crucial to to test the performance of the On Board Computer (OBC), before mounting all the components onto a 1U *CubeSat*.

The aim of this test is to determine how we can control the motor along the Z axis (yaw) and lead the system to the desired position. For this test the assembly of the electronic components of the AOCS system will be placed and tested on a wood *Lazy Susan*[1]. In practice, we will learn how to tune de PID and test the real behaviour of the brushless motor.

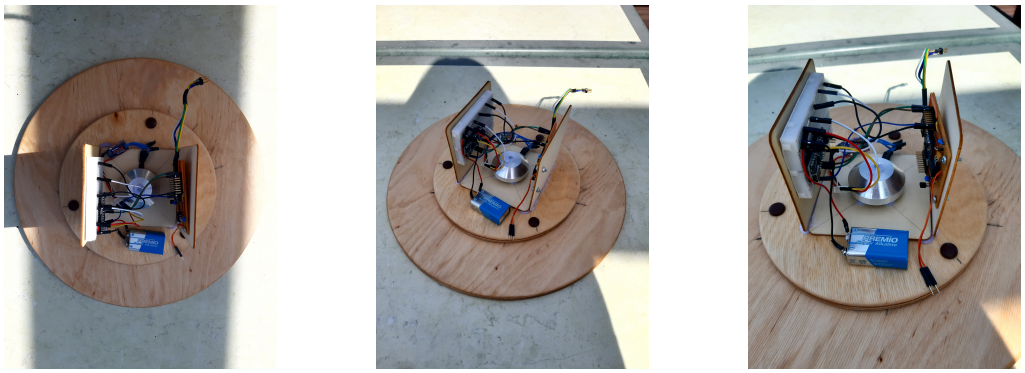


Figure 6.1: Experimental setup for the test of the yaw angle with a *Lazy Susan*

Figure 5.1 shows the set up of the test. The *Lazy Susan* is supposed to spin to achieve the desired setpoint.

The components used for the setup of this test are the following:

- Arduino as OBC: The original choice was the Launchpad CC3200, but because of technical issues that will be described below, we opted for a change to Arduino.
- A 9V battery: a 3.3 V battery was the initial option. However, the change of the OBC to Arduino allowed the use of a single 9V battery, as the accepted input for Arduino is in a range of 6-20V. Avoiding the necessity of an exact voltage input became an additional advantage.
- One reaction wheel.
- One brushless motor.
- One Electronic Speed controller.

[1] A Lazy Susan is a rotating tray placed on a table or countertop to aid in distributing food.

6.1.. Fundamentals of a PID

The PID, that is, the proportional integral derivative controller, is one of the most important parts, is a crucial component in our device. A well designed PID will correct the mass center positioning errors, as well as errors caused by external forces as atmospheric or motor friction. A PID is basically needed to control the position of the *CubeSat* and will be programmed in the launchpad. The fundamental knowledge needed to understand how a PID works will be explained in this section.

The system that we want to control (or whose behaviour we want to affect) is called the plant. The input of the plant is the actuated signal, and its output is the controlled variable. The basic idea of a control system is to figure out how to generate the appropriate actuated signal, the input, so that our system will produce the desired controlled variable, the output. The desired output is called setpoint. In the particular case of this project the setpoint is the position (x,y,z) given in angles. In feedback control, the output of the system is fed back and compared to the command data, in order to see how far off the system is from the position where we want it to be (See figure 6.2). The difference between the actual and desired value is the error term. If the output were exactly what we commanded it to be, then the error would be zero, therefore we aim to have zero error. A PID controller takes this error term and converts it into suitable actuator commands, so that after iterating over time the error is driven to zero.

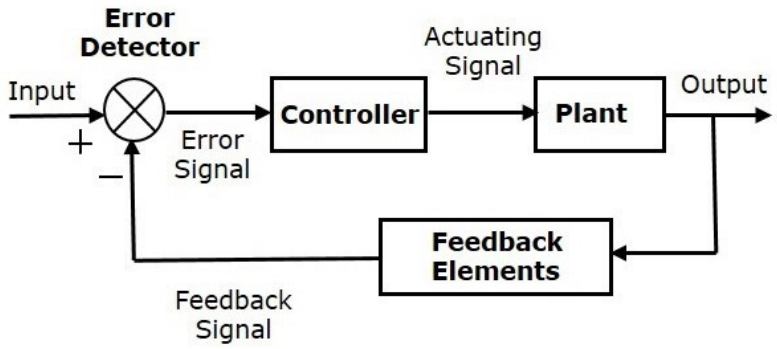


Figure 6.2: Automatic closed loop control system block diagram CREDIT: www.diagram.es

A PID has 3 tuning parameters: Proportional (K_p), Integrator (K_i) and Derivator (K_d). In order to explain the function of each of them we provide the following examples:

- **Proportional controller:** Imagine a person who wants to walk 50 meters along a certain direction. In our example, the person is the plant, the input is the speed and the direction, and the output is the current location. Therefore at the beginning the position error is 50 meters. The brain is the controller and decides how fast to walk. The controller can use the error at the present moment to decide the walking speed. If the controller is set to 0.1, this

means that if we take the error in our system and multiply it by 0.1 (K_p) we will get our walking speed. With a proportional controller like this the error starts to reduce quickly when we are far away, and then gradually slow down as we get closer to our goal. In this way, when we arrive to the 50 meters goal, the error would be zero and the proportional controller would multiple this by 0,1 generating a null speed.

Let us now introduce this proportional controller to a different system: a drone. This time we want a drone fly up to 50 meters, and once that altitude is reached, let the drone start to hover. The new plant of the system now is the drone and the output is the altitude. How would a proportional controller work this time? When the drone is on the ground there is an error of 50 meters, which would generate a large propeller speed, causing the drone to lift off and start to rise. Once the 50 meters are reached, the error would be zero, therefore the speed of the drone would also be zero, causing the drone to fall back to Earth. There is a certain speed where the Lift is equal to the Weight of the drone making it to hover, but the proportional controller is unable to achieve this speed with zero error. No matter which K_p is chosen, the error will only get smaller but never disappears. The schematics of the behaviour of a proportional controller is shown in Figure 6.3.

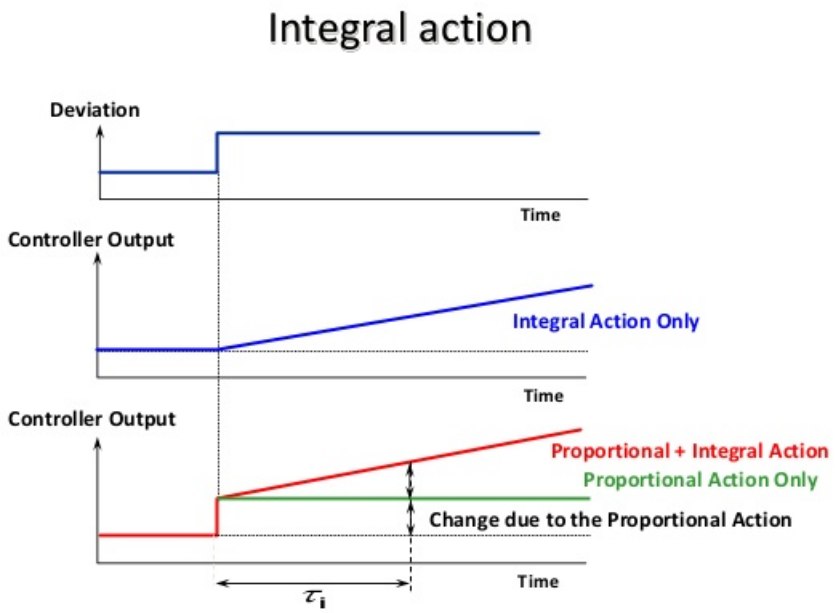


Figure 6.3: Output of a proportional integral controller (red) compared to that of an only proportional (green) and an only integral (blue) controller. CREDIT: www.diagram.es

- **Integrator controller:** The former constant error is called steady state error, and it can be avoided by using past information, specifically adding an integrator path, K_i, alongside the proportional path. An integrator sums up the input signal over time keeping a running total. Therefore, it has memory of what happened before. So if the drone gets to a steady state below the desired altitude, the error term is different to zero, and when integrated,

the output increases. As long as there is error in the system the integral output will rise (See figure 6.3), increasing by its time the speed of the propellers and making the drone to continue rising. Once the desired altitude is reached, the proportional path is doing nothing since the error is zero. Right before the 50 meters high the integral path is making the drone to rise, and when it receives the feedback, it sees that it is at an altitude higher than 50 meters, slowing it down again to a lower altitude. Therefore the integral path is summing and subtracting values all the time to reach the hovering speed. The schematics of the behaviour of an integral controller together with that of a combined proportional integral controller is shown in Figure 6.3.

- **Derivator controller:** The former excess of speed or altitude (negative error) is called overshoot and might not be desired. It can be corrected by adding a derivative path (K_d) to our controller which can extrapolate future positions and respond to how fast we are reaching our goal. A derivative produces a measure of rate of change of the error and how fast it is growing or decreasing. In our example, if the drone is rising quickly, the error is decreasing quickly, therefore it has a negative rate of change which will produce a negative value through the derivative path. This will be added to the controllers output, resulting in a decreasing propeller speed. Basically the controller is predicting how far we are from the goal, and slowing down the propeller to prevent it from overshooting (See figure 6.4).

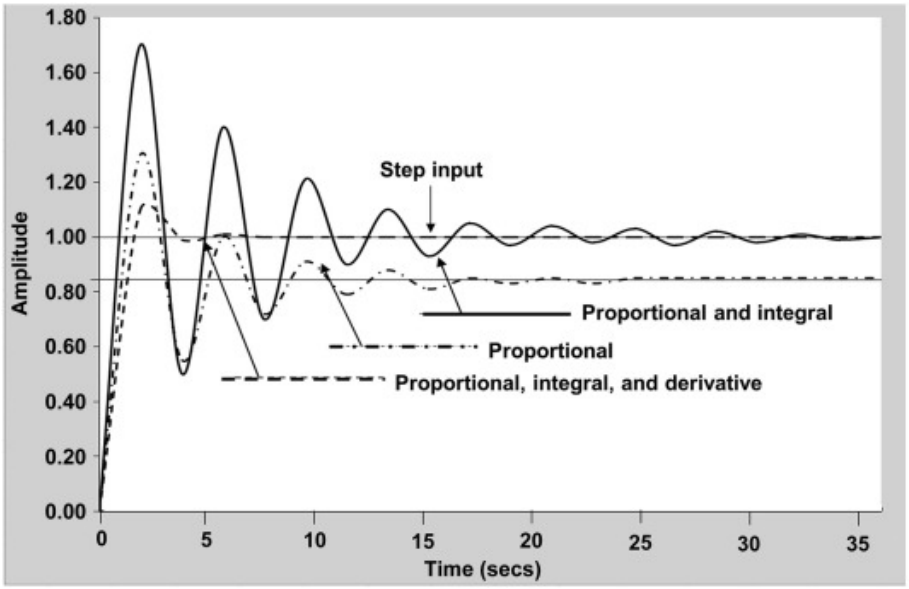


Figure 6.4: Overshoot with different path controllers. CREDIT: sciencedirect.com

Altogether, the PID is a versatile controller that uses the present error, the past error and a prediction of the future error to calculate accurate actuator commands. In this work, the K_p, K_i and K_d parameters will be tuned with a trial and error method.

6.2.. Firmware

As an Arduino is being used for this test, the firmware is done with the Arduino integrated development environment (IDE) in C++.

The libraries used for this connection are:

- *Servo.h*, a library to control the motor.
- *AdafruitBNO055.h* and *AdafruitsensorBNO055.h*: libraries to read the BNO 055 sensor.
- *PID.h*: a proportional integral derivator library.

The motor has both reverse and regular mode. In reverse mode the motor spins counterclockwise and, hence the *Lazy Susan* moves clockwise. On the other hand, in regular mode the motor spins clockwise. The test consisted on setting a setpoint angle in between 0° and 360°. Without configuration, if the *CubeSat* is to rotate from 5° to 355°, it spins always clockwise (reverse motor mode, doing the longest and less efficient path. In order to solve this, a simple code has been implemented and it can be described with the flow chart shown in Figure 6.5. The detailed code is in Appendix A.4.:

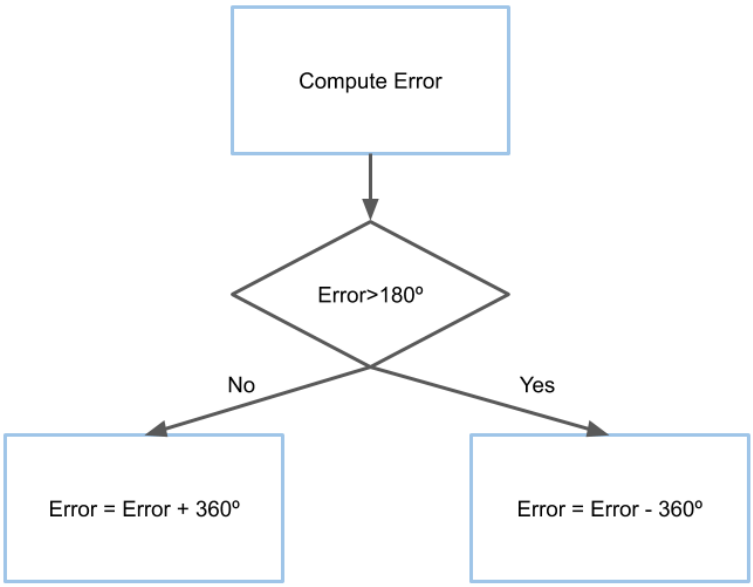


Figure 6.5: Error calculation flowchart.

6.3.. Results of the test

The objective of this test is to control de rotation about the y axis of the system mounted in the *Lazy Susan*. In this test the effect of the position of the center of mass is not taken into account, as we merely aim to study the control of the PID. However, we (visually) center all the elements as much as possible.

Over 60 attempts have been performed with different angles of rotation. Choosing too small or big angles is not optimal. A too small angle 7º makes the performance difficult to observe by sight, as the variations of speed in the motor are almost imperceptible. On the other hand, for too big angles 150º, the performance observed is the same than for smaller angles, but with the added disadvantage of a greater battery consumption. For these reasons, the angle chosen for the test is 30°. In addition, we found by trial and error after several attempts, that the best PID parameters for this system were the following: P_k: 15, P_d: 0, P_i: 0.05. In order to describe the performance of the system, the following parameters are taken into account: speed of the motor, friction in the system, error and time.

- **Friction:** This factor is the most critical in the test, as the work which has to do the motor to achieve the setpoint is proportional to friction. In this test the *Lazy Susan* weights 700 grams, and friction between it and the bearings is very high. The operational *CubeSat* to be launched will not have this problem, as the only significant friction will be due to contact between the engine and its bearings.
- **Speed performance:** When moving from 0° to 30°, the motor starts accelerating to its maximum speed until it is close to the setpoint, then it decelerates until it stops. If friction were not as high as mentioned above, it would not be necessary to accelerate up to maximum speed. In almost zero-friction conditions the acceleration would simply be proportional to the weight of the *CubeSat*.
- **Time:** The time elapsed between the initial position until the setpoint is considerably high. The time needed to rotate from 0° to 30° is approximately 2 minutes, even when the motor is spinning with full speed. Once more, this is due to the high friction of the system in this test environment. Given our experimental setup it could not be decreased nor corrected. The only solution to this would be to build a friction-less spinning platform. However, given the scope of this test, the time is not an important factor as the objective is simply to rotate from an initial position up to the setpoint. As mentioned before, in real conditions, with much smaller friction, the motor would achieve its desired position in a reasonable time (few seconds).
- **Final position error:** High friction also affects the time of the response of the PID. This means that it reaches approximately 38° before it becomes aware that it overshot the desired final position. As a consequence, the system moves back and forth several times until it reaches 30°.

In conclusion, friction has been the most determining factor for this first test, and has affected negatively the performance of the system. Anyway the tuning of the PID has helped to correct this up to a certain point.

6.3.1.. Problems during the test

- During the test of the control in the yaw angle two launchpads where used, and both of them failed due to unexpected overheating and code compil-

ing problems. The implemented solution was to change the OBC for an Arduino. Furthermore, it has been observed in several occasions that the Launchpad CC3200 shows a poor behaviour which can compromise the space missions. Therefore, we strongly recommend to find an reliable alternative to this model.

- We detected that the system was rotating mainly because of the vibration of the motor rather than due to momentum conservation, as the rotation of the system was independent of the rotation of the motor. Basically it was rotating on a different direction than it should.

- Another problem found in this test is the poor performance of the engine for low values of power, which will be described in detail in the following section.

CHAPTER 7. PERFORMANCE TEST OF THE CONTROL IN THE YAW ANGLE

In this Chapter we describe the major changes in the system which were implemented in the second test. These changes were aimed to correct the problems of the qualitative test described in Chapter 6, by reducing friction and replacing the OBC. Finally, we present the data of the performance of the PID.

7.1.. Changes in the experimental setup

The main alterations in the experimental setup can be summarized as follows:

- **Changes in the test bed:**

  The new test bed consists of a bearing attached on one side by a wooden base and on the other side by the CubeSat (See figure: 7.1). This test bed is lighter and has less friction due to the bearing, therefore the behaviour of the CubeSat is less condition by friction.

Figure 7.1: Wooden base with an attached bearing.

- **Changes in the OBC:**

  The Launchpad CC3200 showed a poor performance during the first test, therefore it has been replaced with a NODEMCU(See figure: 7.2). This MCU has Wi-Fi, which will be used to obtain the data of the sensor and plot it. It is not an official Arduino and was only used for testing purposes, as a replacement of the original Launchpad.

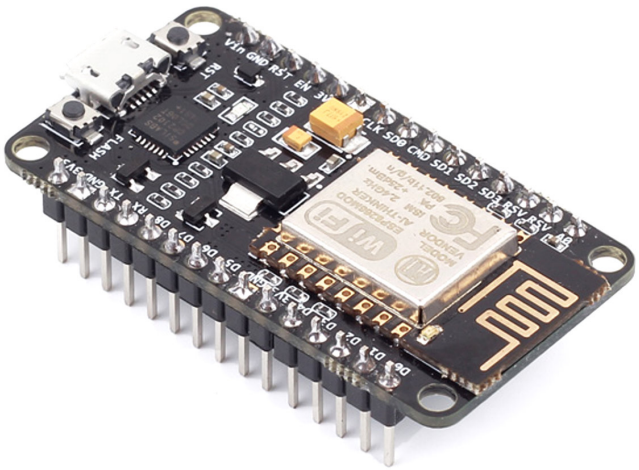


Figure 7.2: NodeMCU

- **Changes in the Firmware:**

  We observed during the first test that the engine had to spin at full speed if a minimum angle variation was to be detected. Once the engine reached maximum speed and ω, the system would slightly move due to the vibration of the engine, rather than due to momentum conservation. Most of the rotation was done at the beginning, when the engine was accelerating. Thus, for this second test, we decided to let the engine reach the speed calculated by the PID, then let it stop, and repeat the process until the setpoint was reached. This allowed the system rotate every time the engine accelerated and reach faster the setpoint. Note that this solution is a temporary workaround for our current test bed, and will not be needed for the operational CubeSat.

7.2.. New firmware

The connection with the engine works as follows: The Arduino sends PWM signals to the ESC with values in between 1000 and 2000 microseconds. The ESC translates that signal to the engine, but the value in Watts can not be known, thus the Power is expressed in microseconds. There have been made two changes regarding the engine performance. The servo has been configured as follows:

- Reverse engine mode: Signal from $1000\,\mu s$ to $1500\,\mu s$.
- Clockwise engine mode: Signal from $1500\,\mu s$ to $2000\,\mu s$.

If the signal is at $1500\,\mu s$, the engine does not spin. Actually, on the first test, we observed that this engine does not perform adequately for low power values. Whenever the power would be in between 1400-1600 the engine would try to spin with a high difficulty. Usually it would stop working after that, which in a real mission would be a major drawback. In order to solve this the engine is forced to

avoid the intervals of power from 1400 to 1600. Furthermore as described in the previous subsection, the engine will accelerate and stop for each value of power given by the PID in order to counteract the effect of friction. The code imple-

mented is described with the following flowchart:The detailed code is in Appendix A A.1..

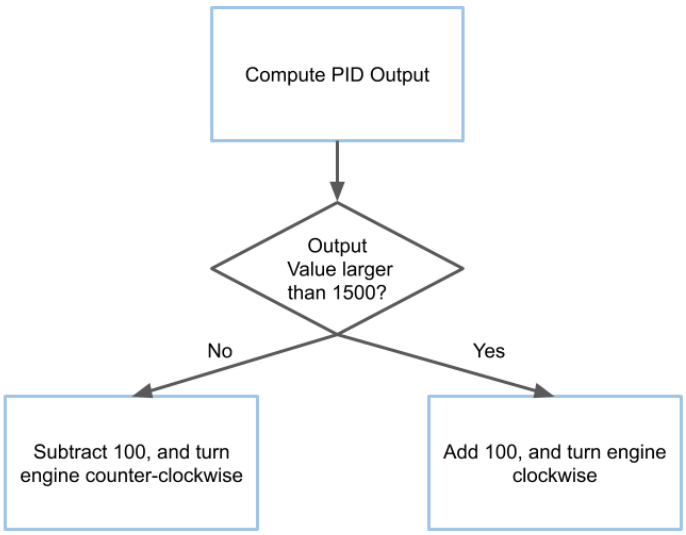


Figure 7.3: Engine rotation according to PID output flow chart.

In order to obtain the data from the sensor in real time, it has been used an API. The data is uploaded to it and afterwards extracted and shown in the terminal. This implementation has been done in C, and can be seen in Appendix A.1..

Afterwards in order to obtain the data from the code has been done in Python.

The flowchart of the entire code can be seen in figure 7.4

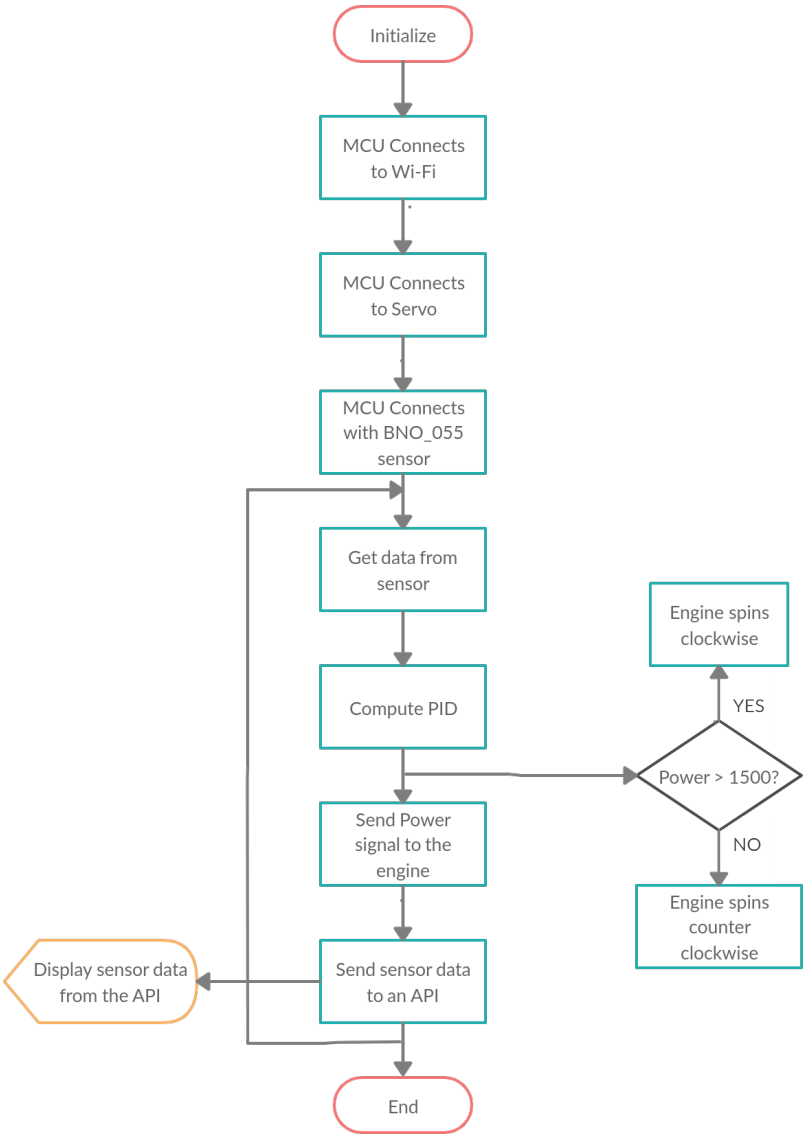


Figure 7.4: FlowChart of the OBC Firmware

7.3.. Performance of the PID for different tuning parameters

After confirming the new test bed is robust and stable, we performed several tests for different tuning parameters of the PID. The objective of this is to observe the behaviour of the system when varying the aforementioned tuning parameters. For this test the setpoint is 300º.

All the tests have been stopped manually when the setpoint was reached, in order to prevent the overheating of the components, as well as discharging the battery. This is the main cause why the plots shown next do not have a clear 0 error when time tends to infinity. Ideally, the system should have been observed for a longer time, until the error achieved reached values closer to zero.

7.3.1.. Variation of K_p

For this test we keep the values of K_i and K_d constant, and vary the values of the K_p. When changing the K_p value, the behaviour is quite clear: the higher K_p, the higher the absolute value of the power output of the motor. However, in figure 7.5 we can see that the time to reach the set point is quite similar in all cases (the case for 0.8 could be considered an outlier). This is probably due to effect of friction, a higher K_p is able to move the system faster for large error values, but as it reaches the setpoint, the effect of the Proportional controller is not strong enough to overcome friction, leaving the system dependent on the Integrator controller.

7.3.2.. Variation of K_i

Figure 7.6 shows the effect of varying K_i while keeping constant $K_p = 0.8$ and $K_d = 0.03$. Note that the behaviour of the angle is very similar for the lower values of K_i (=1 and 3.5), although it takes a much longer time to reach the setpoint value. As K_i increases, the system starts oscillating with amplitudes and frequencies which increase with K_i (see upper panel). Correspondingly, the proxy for the power presents variations in order to account for the oscillations (see lower panel). On the other hand, despite the oscillations and the overshooting, the system is able to reach faster the setpoint when high K_i values are used. The final design should consider a compromise solution between the time to reach the setpoint, the power consumption, and the eventual problems which these type of oscillations might cause.

7.3.3.. Variation of K_d

The change in K_d is crucial in this system. For small values of K_d, there is a large overshoot and the system starts oscillating around the setpoint (see figure: 7.7). With a high value of K_d (0.5 or 1), the controller is able to brake and prevents the overshoot. Given the importance of the proportional controller in this system, the k_d is also very important to prevent the overshoot that the k_p can cause.

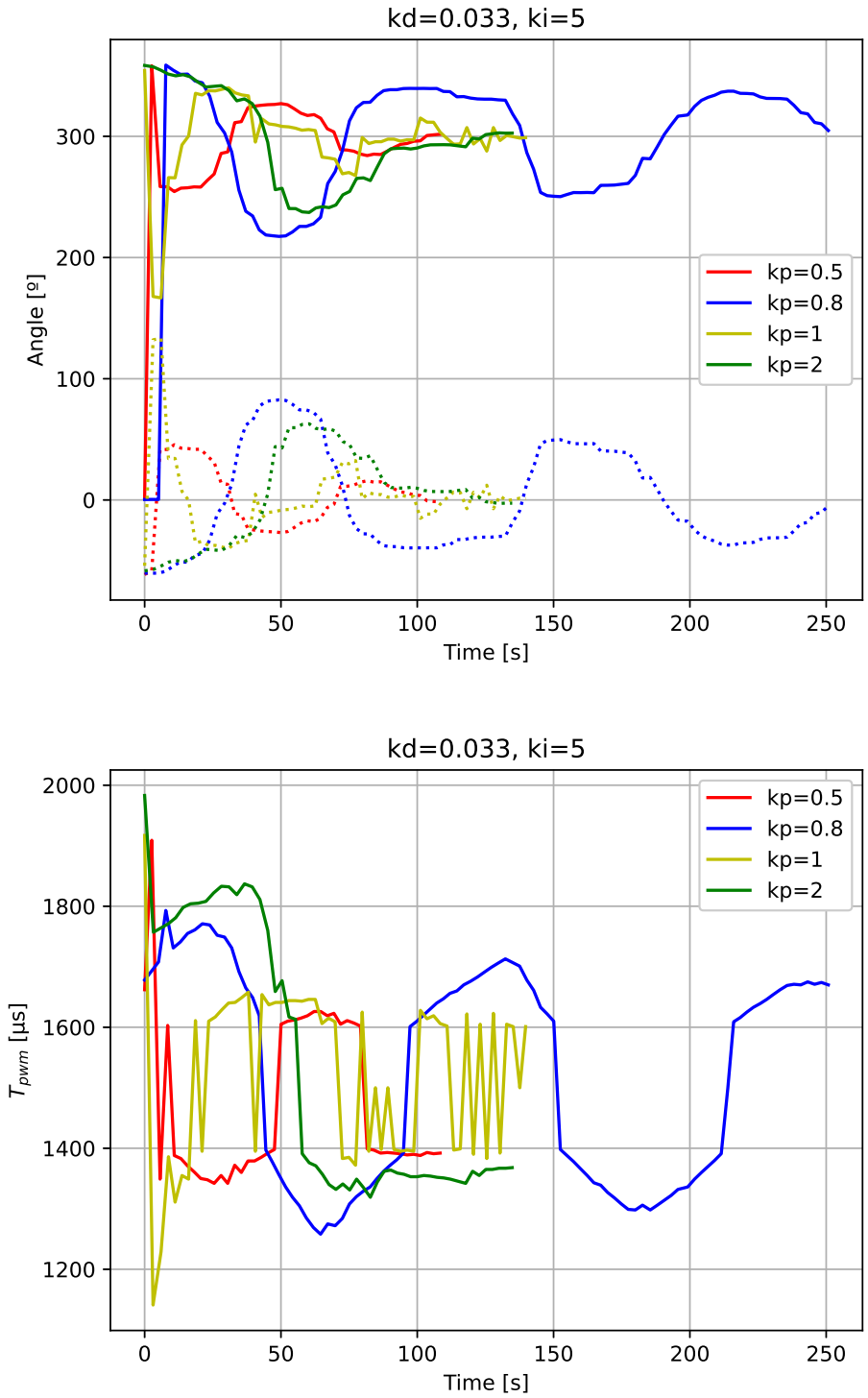


Figure 7.5: Sensor output data for K_i = 5, K_d = 0.033 and variable K_p = 0.5, 0.8, 1 and 2. Upper panel shows the angle (solid lines) and the error (dotted lines) versus time. Lower panel shows the time of the PWM signal, which is a proxy for the power supplied, versus the time

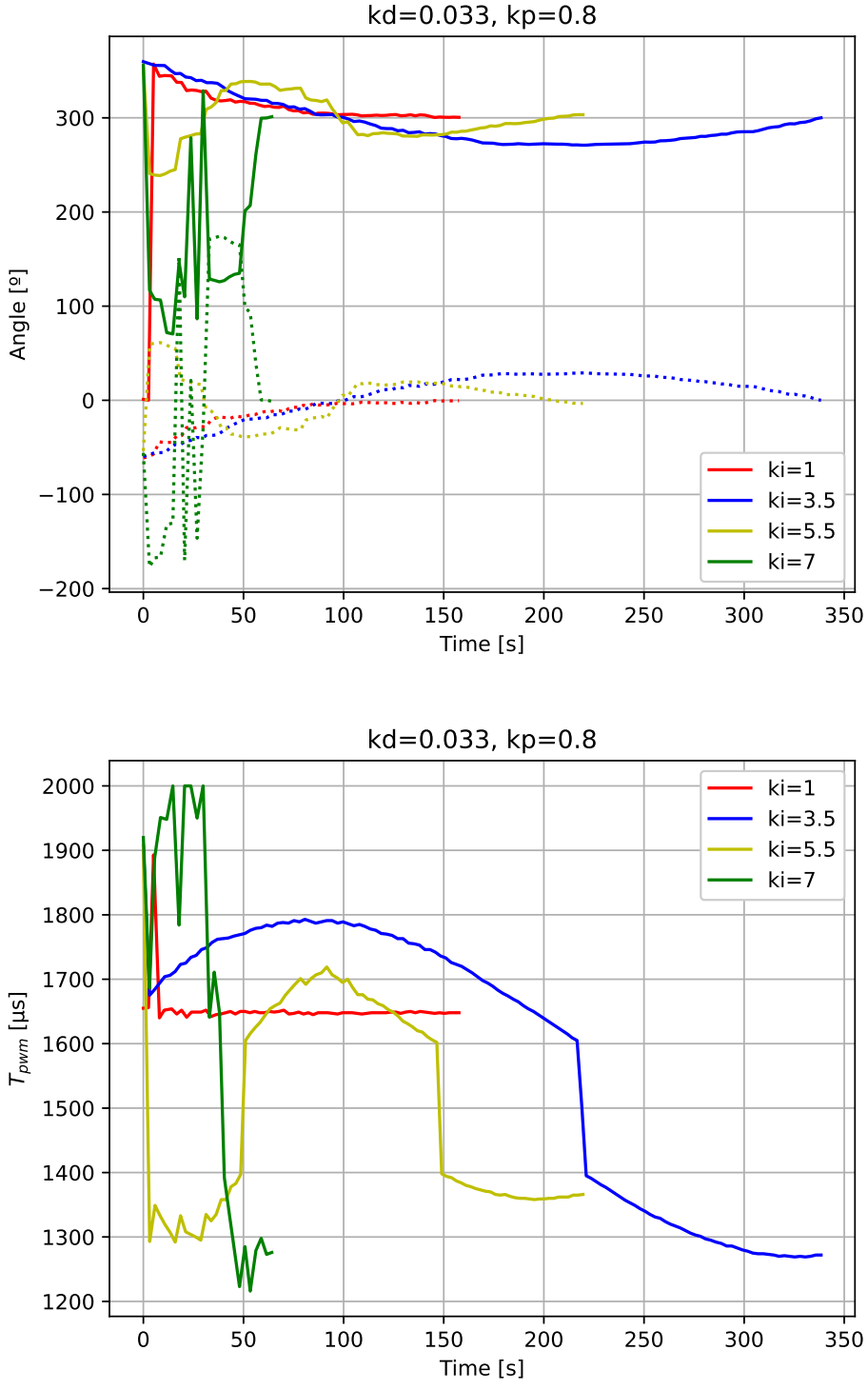


Figure 7.6: Sensor output data for K_p = 0.8, K_d = 0.033 and variable K_i = 1, 3.5, 5.5 and 7. Upper panel shows the angle (solid lines) and the error (dotted lines) versus time. Lower panel shows the time of the PWM signal, which is a proxy for the power supplied, versus the time

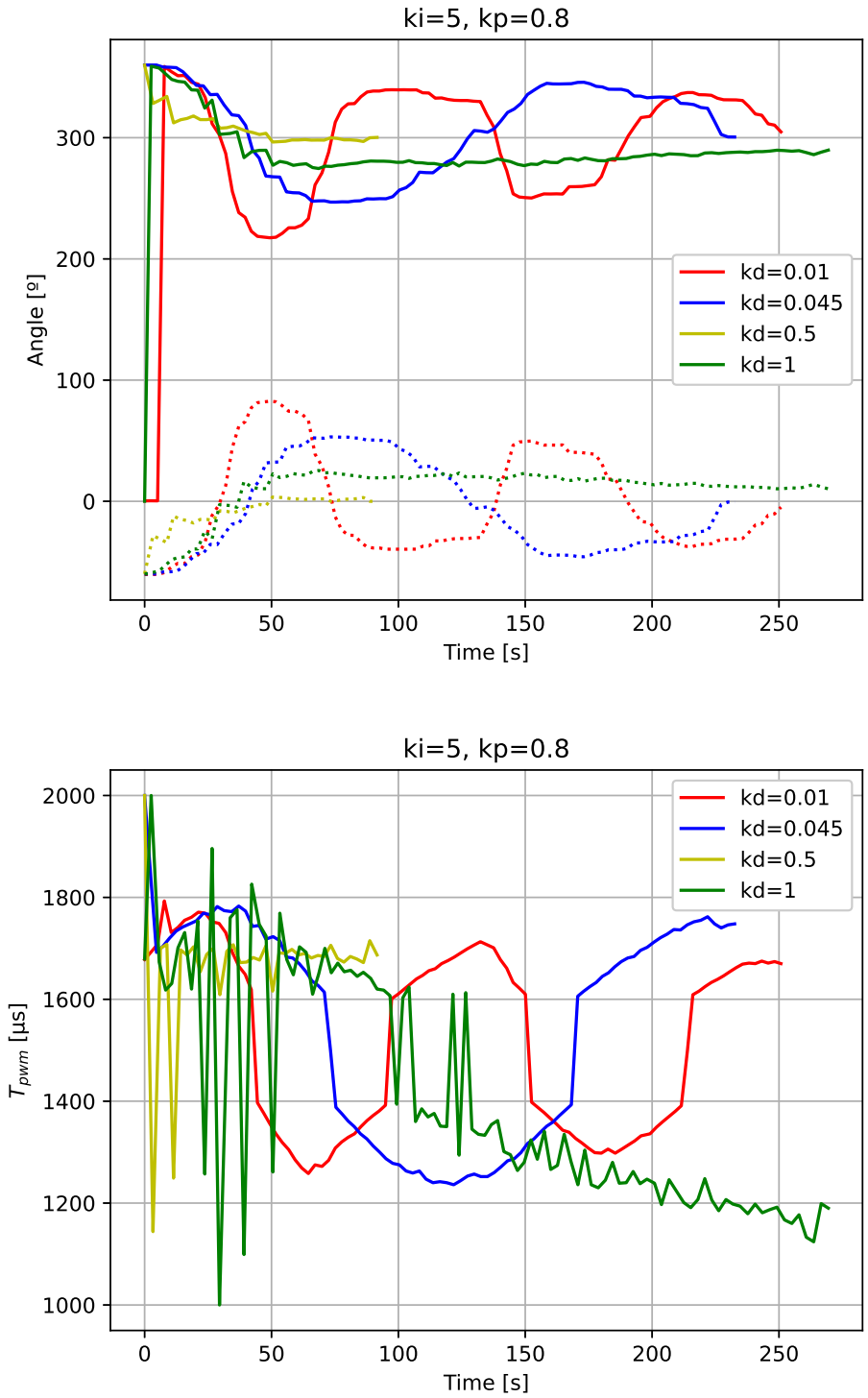


Figure 7.7: Sensor output data for K_p = 0.8, K_i = 5 and variable K_d = 0.01, 0.045, 0.5 and 1. Upper panel shows the angle (solid lines) and the error (dotted lines) versus time. Lower panel shows the time of the PWM signal, which is a proxy for the power supplied, versus the time

7.4.. Performance of the PID for different setpoints

The performance of the PID for a small value of setpoint (10º) and a larger one (180º) was also tested (see Figures 7.8 to 7.9).

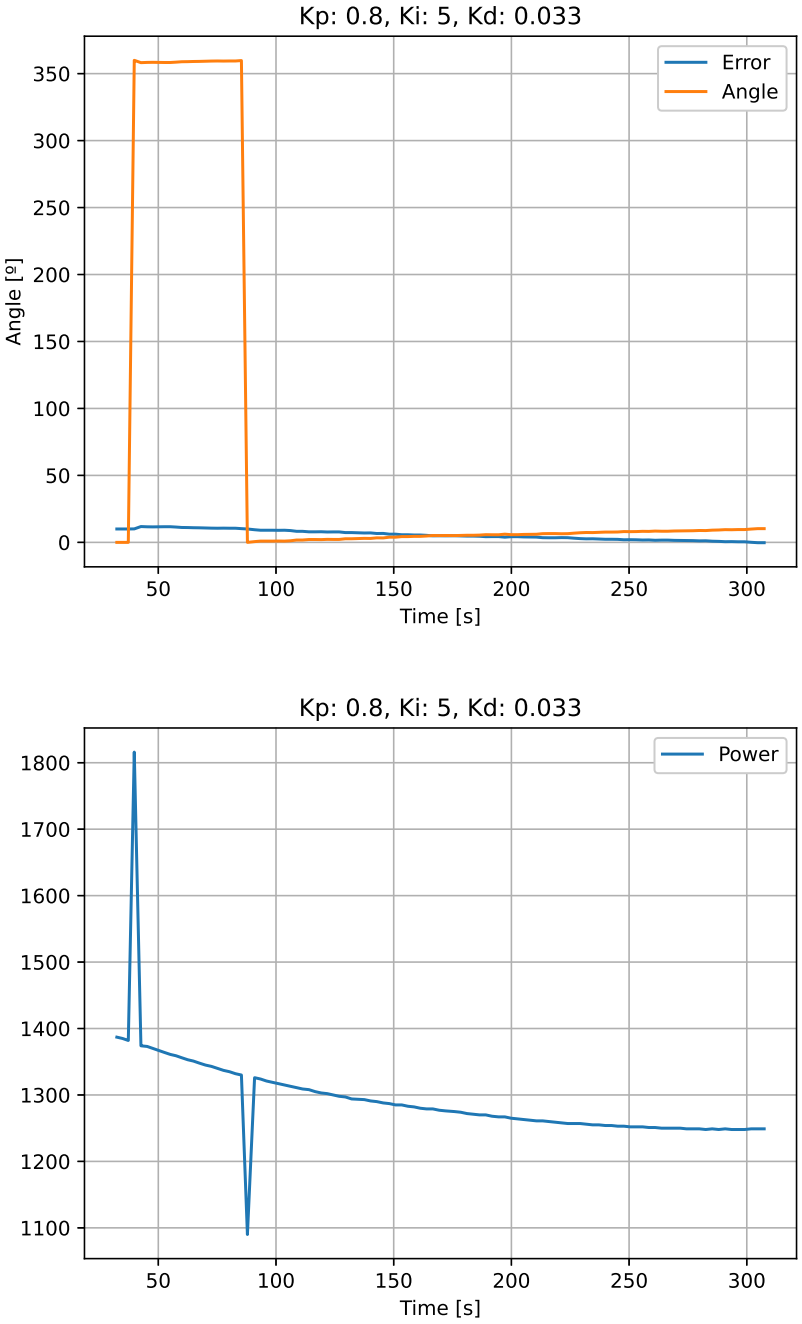


Figure 7.8: Sensor data 10º setpoint

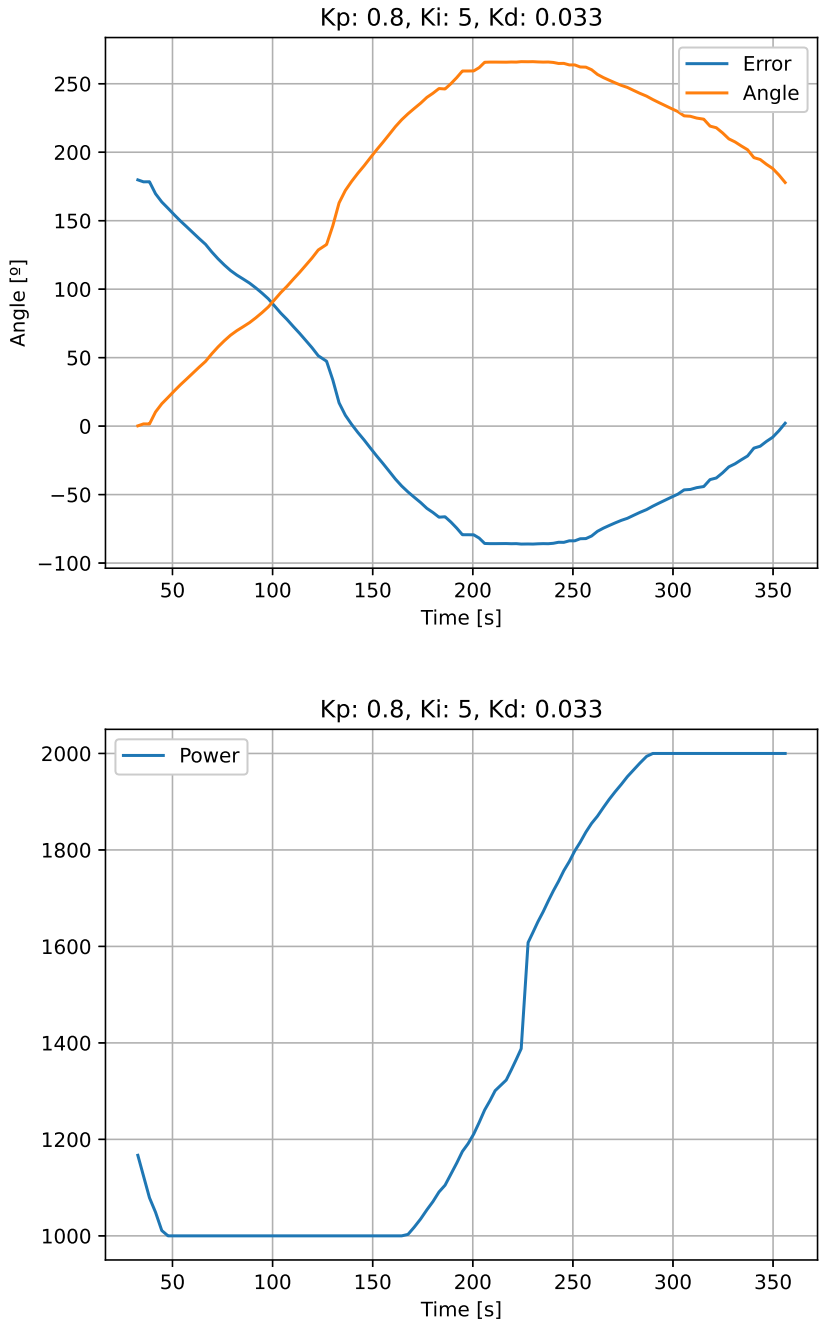


Figure 7.9: Sensor data 180º setpoint

The behaviour of the PID is equally correct for small and large angles. As shown in Figures 7.8 to 7.9, the time is not linearly proportional to the angle of the setpoint. In fact, for a rotation of 10º, the system barely moves due to friction, because the power of the engine computed by the PID is very low.

7.5.. Expected results vs obtained results

The summary of expected results versus obtained results is presented in Table 7.1.

Table 7.1: Summary of expected results versus obtained results.

Expected performance	**Observed performance**
The power of the engine is proportional to the K_p.	The general trend is that the power is higher for higher values of K_p, due to the effect of friction.
For high values of K_i the overshoot is also high.	The higher the value of K_i, the sooner the system arrives to the setpoint, although it also overshoots a larger angle.
For high values of K_d, the system decelerates faster the faster the setpoint is being approached.	In general, the higher the value of K_d, the lower the overshoot angle, and the higher the deceleration required.
The PID performance does not depend on setpoint values.	The PID works properly both for large and small setpoint values.
Friction can be overcome with a good tuning of the PID.	The effect of friction causes an unpredictable behaviour in the system, but in general it can be corrected by the PID. The integrator is the most important tuning parameter in a system with friction.
The engine performs correctly for any power value.	The engine works well for high power values, but for low values (1400 µs and 1600 µs) the engine presents unexpected behaviours (sometimes it stops) and is not able to perform correctly.
The system performs correctly for every value of power	The system shows poor performance for low values due to the negative effect of friction.
The new test bed has less friction than the original one.	The bearing has considerably less friction than the *Lazy Susan*, which helps improve results.

CHAPTER 8. CONCLUSIONS AND FUTURE WORK

8.1.. Summary and conclusions

In this project we were able to thoroughly analyze the behaviour of a PID.

i) We have analysed the behaviour of a simplified version of the an Attitude Orbit Control System (AOCS) in one axis. Over 100 tests were carried out for different tuning parameters and setpoints in order to do a gap analysis of the system.

ii) Regarding the components of the system, all the chosen elements have been tested during months in order to see their performance. All the elements except the launchpad showed a good performance during the test, although it has flight heritage and has proven to work well in real space missions.

iii) The reaction wheels used displayed significant vibration, which could affect the rotation of the system. Note that, originally, the test was going to be performed with better reaction wheels, actually with models which had flight heritage, but due to COVID-19 restrictions it was not possible.

iv) We have performed a quantitative exploration of the effects of altering the proportional integrator derivator parameters K_p, K_i and K_d. Although the conditions of the real CubeSat will be different from those of our simplified experimental setup, information can be eventually useful as a guideline.

v) These were the two main problems during the tests, together with the friction of the test bed, but luckily these inconveniences will disappear when on site tests are possible again.

The development of this work was deeply influenced by the effects of the COVID-19 restrictions. The original scope of the project had to be severely adjusted and adapted to the resources available, as in-lab tests were not possible.

8.2.. Future work

The work done so far is the first part of this project. Now we propose possible lines of further work that should be implemented in order to successfully complete the project.

i) CATIA Model: Once the final components of the AOCS are determined, a CATIA 3D model should be developed in order to calculate the inertia tensor, and the span rotation matrix of the system (which can be converted into a quaternion vector).

ii) PID for the 3 axis. The code developed for this thesis must be extended to 3 axis rotations, using the data provided by the CATIA model.

iii) The 6 faces of the CubeSat should be printed in 3D in order to accommodate the AOCS system.

iv) Tests in a special test bed at ESAC should be performed. In order to understand the details of the behaviour of the system in a full 3 axis rotation, an environment with almost zero friction is required. In order to achieve this, an compressed-air test bed should be used.

v) The definitive laboratory tests must ensure that the system performance and output values meet the actual requirements of the mission. An optimal compromise solution involving reasonably quick system response, stability and low energy consumption will be attained.

BIBLIOGRAPHY

[1] Nanosats database. https://www.nanosats.eu/.

[2] Launchpad user manual. http://www.ti.com/lit/ug/swru372c/swru372c.pdf.

[3] Hobby king brushless motor specs. https://hobbyking.com/es_es/multistar-elite-2204-2300kv-multi-rotor-motor-cw-prop-adapter.html.

[4] Hobby king electronic speed controller user manual. https://cdn-global-hk.hobbyking.com/media/file/407699232X4683562X51.pdf.

[5] Bno055 sensor. https://cdn-shop.adafruit.com/datasheets/BST_BNO055_DS000_12.pdf.

APPENDICES

APPENDIX A. OBC CODE

A.1.. Main PID Code

```
#include <Wire.h>
//#include <BMA222.h>
#include <Servo.h>
#include "PID.h"
#include "Adafruit_Sensor.h"
#include "Adafruit_BNO055.h"
#include "utility/imumaths.h"

// libraries to connect with the API
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WebServer.h>
#include <ESP8266HTTPClient.h>

//Set the credential of the local WI-Fi
const char *ssid = "MiFibra-8DAE";
const char *password = "9cYQg59X";

//Web/Server address to read/write from
const char *host = "192.168.1.123:5002";
//BMA222 mySensor;
const int highestPin = 29;
int value, value_prev(1200);
Servo myservo;
int pin_servo = 16;
int val;
unsigned long time_now;
unsigned long time_before;
int interval = 1000;
double Setpoint(60), Input, Output;
Adafruit_BNO055 bno = Adafruit_BNO055(55);

//Specify the links and initial tuning parameters
double Kp = 0.8, Ki = 5, Kd = 0.033, outmax=500, outmin=-500;
PID myPID(&Input, &Output, &Setpoint, Kp, Ki, Kd, DIRECT);


void setup()
{
  delay(1000);
  Serial.begin(115200);
  WiFi.mode(WIFI_OFF);      //Prevents reconnection issue (taking too long
     to connect)
  delay(1000);
```

```
 WiFi.mode(WIFI_STA);      //This line hides the viewing of ESP as wifi
     hotspot

 WiFi.begin(ssid, password);  //Connect to your WiFi router
 Serial.println("");

 Serial.print("Connecting");
 // Wait for connection
 while (WiFi.status() != WL_CONNECTED) {
   delay(500);
   Serial.print(".");
 }

//If connection successful show IP address in serial monitor
 Serial.println("");
 Serial.print("Connected to ");
 Serial.println(ssid);
 Serial.print("IP address: ");
 Serial.println(WiFi.localIP());
 myservo.attach(pin_servo);
 myservo.writeMicroseconds(2000);
 delay(5000);
 myservo.writeMicroseconds(1000);
 delay(5000);
 myservo.writeMicroseconds(1500);
 Serial.println("Setting up servo");
 delay(5000);
 myservo.writeMicroseconds(1500);
 Serial.print("The setpoint is: ");
 myPID.SetOutputLimits(outmin, outmax);
 myPID.SetMode(AUTOMATIC);
 myPID.SetSampleTime(50);
 //pinMode(highestPin, OUTPUT);

 /* Initialise the BNO 055 sensor */
  if(!bno.begin())
 {
   /* There was a problem detecting the BNO055 ... check your
      connections */
   Serial.print("Ooops, no BNO055 detected ... Check your wiring or I2C
      ADDR!");
   while(1);
 }
 Serial.println("After BNO begin");
 delay(1000);

 bno.setExtCrystalUse(true);

 /*Define variable time_before, that corresponds to milliseconds from
     the start of the programm */
```

```
  time_before = millis();

}

void loop()
{
  time_now = millis();

  /* Get a new sensor event */
  sensors_event_t event;
  bno.getEvent(&event);

  // int8_t data = mySensor.readXData();
  Input = event.orientation.x;
  myPID.Compute();
  value = map(-Output,-500,500,1000,2000); //Map the output to the
      corresponding values of the ESC
 if (value > 1500){
    value = min(2000,value+100);
    for (int i = 1500; i<value; i=i+50){
    myservo.writeMicroseconds(i);
    delay(100);
  }
 }
 if (value < 1500){
    value = max(1000,value-100);
    for (int i = 1500; i>value; i=i-50){
    myservo.writeMicroseconds(i);
    delay(100);
  }
 }
  myservo.writeMicroseconds(value);
  delay(100);
  myservo.writeMicroseconds(1500);
  delay(2000);

  value_prev= value; //Compute time difference to check if API update is
      needed
  if (time_now - time_before > interval) {
  HTTPClient http;  //Declare object of class HTTPClient
  String ADCData, station;
  char postData [100];

  //Post Data

  sprintf(postData,"{\"power\": %d, \"angle\":%f, \"time\":%lu,
      \"error\":%f}", value,event.orientation.x,time_now,
      myPID.error_PID);
  //Serial.println(postData);
  http.begin("http://192.168.1.123:5002/food/upload");        //Specify
      request destination
```

```
  Serial.println(postData);
  http.addHeader("Content-Type", "application/json"); //Specify
      content-type header
  int httpCode = http.POST(postData); //Send the request
  http.end(); //Close connection
  time_before = time_now;
  }

}
```

A.2.. Adafruit BNO 055 Library

```
/*!
 * @file Adafruit_BNO055.cpp
 *
 *  @mainpage Adafruit BNO055 Orientation Sensor
 *
 *  @section intro_sec Introduction
 *
 *  This is a library for the BNO055 orientation sensor
 *
 *  Designed specifically to work with the Adafruit BNO055 Breakout.
 *
 *  Pick one up today in the adafruit shop!
 *  ------> https://www.adafruit.com/product/2472
 *
 *  These sensors use I2C to communicate, 2 pins are required to
     interface.
 *
 *  Adafruit invests time and resources providing this open source code,
 *  please support Adafruit andopen-source hardware by purchasing products
 *  from Adafruit!
 *
 *  @section author Author
 *
 *  K.Townsend (Adafruit Industries)
 *
 *  @section license License
 *
 *  MIT license, all text above must be included in any redistribution
 */

#include "Arduino.h"

#include <limits.h>
#include <math.h>

#include "Adafruit_BNO055.h"
```

```
/*!
 *  @brief  Instantiates a new Adafruit_BNO055 class
 *  @param  sensorID
 *          sensor ID
 *  @param  address
 *          i2c address
 *  @param  theWire
 *          Wire object
 */
Adafruit_BNO055::Adafruit_BNO055(int32_t sensorID, uint8_t address,
                                 TwoWire *theWire) {
  _sensorID = sensorID;
  _address = address;
  _wire = theWire;
}

/*!
 *  @brief  Sets up the HW
 *  @param  mode
 *          mode values
 *           [OPERATION_MODE_CONFIG,
 *            OPERATION_MODE_ACCONLY,
 *            OPERATION_MODE_MAGONLY,
 *            OPERATION_MODE_GYRONLY,
 *            OPERATION_MODE_ACCMAG,
 *            OPERATION_MODE_ACCGYRO,
 *            OPERATION_MODE_MAGGYRO,
 *            OPERATION_MODE_AMG,
 *            OPERATION_MODE_IMUPLUS,
 *            OPERATION_MODE_COMPASS,
 *            OPERATION_MODE_M4G,
 *            OPERATION_MODE_NDOF_FMC_OFF,
 *            OPERATION_MODE_NDOF]
 *  @return true if process is successful
 */
bool Adafruit_BNO055::begin(adafruit_bno055_opmode_t mode) {
#if defined(ARDUINO_SAMD_ZERO) && (_address == BNO055_ADDRESS_A)
#error                                                                         \
    "On an arduino Zero, BNO055's ADR pin must be high. Fix that, then
       delete this line."
  _address = BNO055_ADDRESS_B;
#endif
  Serial.println("Started the sensor");
  /* Enable I2C */
  _wire->begin();
  Serial.println("After the wire begin");
  // BNO055 clock stretches for 500us or more!
#ifdef ESP8266
  _wire->setClockStretchLimit(1000); // Allow for 1000us of clock
     stretching
```

```
#endif
  Serial.println("Before read address");
  /* Make sure we have the right device */
  uint8_t id = read8(BNO055_CHIP_ID_ADDR);
  Serial.println(id);
  Serial.println(BNO055_ID);
  if (id != BNO055_ID) {
    delay(1000); // hold on for boot
    id = read8(BNO055_CHIP_ID_ADDR);
    if (id != BNO055_ID) {
      return false; // still not? ok bail
    }
  }
 Serial.println("Setting Mode");
  /* Switch to config mode (just in case since this is the default) */
  setMode(OPERATION_MODE_CONFIG);
  Serial.println("Write8");
  /* Reset */
  write8(BNO055_SYS_TRIGGER_ADDR, 0x20);
  /* Delay incrased to 30ms due to power issues
     https://tinyurl.com/y375z699 */
  Serial.println("Before while");
  delay(3000);
  while (read8(BNO055_CHIP_ID_ADDR) != BNO055_ID) {
    delay(100);
  }
  delay(5000);
  Serial.println("Got after the while");
  /* Set to normal power mode */
  write8(BNO055_PWR_MODE_ADDR, POWER_MODE_NORMAL);
  delay(10);

  write8(BNO055_PAGE_ID_ADDR, 0);

  /* Set the output units */
  /*
  uint8_t unitsel = (0 << 7) | // Orientation = Android
                    (0 << 4) | // Temperature = Celsius
                    (0 << 2) | // Euler = Degrees
                    (1 << 1) | // Gyro = Rads
                    (0 << 0); // Accelerometer = m/s^2
  write8(BNO055_UNIT_SEL_ADDR, unitsel);
  */

  /* Configure axis mapping (see section 3.4) */
  /*
  write8(BNO055_AXIS_MAP_CONFIG_ADDR, REMAP_CONFIG_P2); // P0-P7, Default
     is P1
  delay(10);
  write8(BNO055_AXIS_MAP_SIGN_ADDR, REMAP_SIGN_P2); // P0-P7, Default is
     P1
```

```
  delay(10);
  */

  write8(BNO055_SYS_TRIGGER_ADDR, 0x0);
  delay(10);
  /* Set the requested operating mode (see section 3.3) */
  setMode(mode);
  delay(20);

  return true;
}

/*!
 *  @brief Puts the chip in the specified operating mode
 *  @param mode
 *         mode values
 *          [OPERATION_MODE_CONFIG,
 *           OPERATION_MODE_ACCONLY,
 *           OPERATION_MODE_MAGONLY,
 *           OPERATION_MODE_GYRONLY,
 *           OPERATION_MODE_ACCMAG,
 *           OPERATION_MODE_ACCGYRO,
 *           OPERATION_MODE_MAGGYRO,
 *           OPERATION_MODE_AMG,
 *           OPERATION_MODE_IMUPLUS,
 *           OPERATION_MODE_COMPASS,
 *           OPERATION_MODE_M4G,
 *           OPERATION_MODE_NDOF_FMC_OFF,
 *           OPERATION_MODE_NDOF]
 */
void Adafruit_BNO055::setMode(adafruit_bno055_opmode_t mode) {
  _mode = mode;
  write8(BNO055_OPR_MODE_ADDR, _mode);
  delay(30);
}

/*!
 *  @brief Changes the chip's axis remap
 *  @param remapcode
 *         remap code possible values
 *         [REMAP_CONFIG_P0
 *          REMAP_CONFIG_P1 (default)
 *          REMAP_CONFIG_P2
 *          REMAP_CONFIG_P3
 *          REMAP_CONFIG_P4
 *          REMAP_CONFIG_P5
 *          REMAP_CONFIG_P6
 *          REMAP_CONFIG_P7]
 */
void Adafruit_BNO055::setAxisRemap(
    adafruit_bno055_axis_remap_config_t remapcode) {
```

```
  adafruit_bno055_opmode_t modeback = _mode;

  setMode(OPERATION_MODE_CONFIG);
  delay(25);
  write8(BNO055_AXIS_MAP_CONFIG_ADDR, remapcode);
  delay(10);
  /* Set the requested operating mode (see section 3.3) */
  setMode(modeback);
  delay(20);
}

/*!
 *  @brief  Changes the chip's axis signs
 *  @param  remapsign
 *          remap sign possible values
 *          [REMAP_SIGN_P0
 *           REMAP_SIGN_P1 (default)
 *           REMAP_SIGN_P2
 *           REMAP_SIGN_P3
 *           REMAP_SIGN_P4
 *           REMAP_SIGN_P5
 *           REMAP_SIGN_P6
 *           REMAP_SIGN_P7]
 */
void Adafruit_BNO055::setAxisSign(adafruit_bno055_axis_remap_sign_t
    remapsign) {
  adafruit_bno055_opmode_t modeback = _mode;

  setMode(OPERATION_MODE_CONFIG);
  delay(25);
  write8(BNO055_AXIS_MAP_SIGN_ADDR, remapsign);
  delay(10);
  /* Set the requested operating mode (see section 3.3) */
  setMode(modeback);
  delay(20);
}

/*!
 *  @brief  Use the external 32.768KHz crystal
 *  @param  usextal
 *          use external crystal boolean
 */
void Adafruit_BNO055::setExtCrystalUse(boolean usextal) {
  adafruit_bno055_opmode_t modeback = _mode;

  /* Switch to config mode (just in case since this is the default) */
  setMode(OPERATION_MODE_CONFIG);
  delay(25);
  write8(BNO055_PAGE_ID_ADDR, 0);
  if (usextal) {
    write8(BNO055_SYS_TRIGGER_ADDR, 0x80);
```

```
  } else {
    write8(BNO055_SYS_TRIGGER_ADDR, 0x00);
  }
  delay(10);
  /* Set the requested operating mode (see section 3.3) */
  setMode(modeback);
  delay(20);
}

/*!
 *   @brief Gets the latest system status info
 *   @param system_status
 *          system status info
 *   @param self_test_result
 *          self test result
 *   @param system_error
 *          system error info
 */
void Adafruit_BNO055::getSystemStatus(uint8_t *system_status,
                                      uint8_t *self_test_result,
                                      uint8_t *system_error) {
  write8(BNO055_PAGE_ID_ADDR, 0);

  /* System Status (see section 4.3.58)
     0 = Idle
     1 = System Error
     2 = Initializing Peripherals
     3 = System Iniitalization
     4 = Executing Self-Test
     5 = Sensor fusio algorithm running
     6 = System running without fusion algorithms
   */

  if (system_status != 0)
    *system_status = read8(BNO055_SYS_STAT_ADDR);

  /* Self Test Results
     1 = test passed, 0 = test failed

     Bit 0 = Accelerometer self test
     Bit 1 = Magnetometer self test
     Bit 2 = Gyroscope self test
     Bit 3 = MCU self test

     0x0F = all good!
   */

  if (self_test_result != 0)
    *self_test_result = read8(BNO055_SELFTEST_RESULT_ADDR);

  /* System Error (see section 4.3.59)
```

```
     0 = No error
     1 = Peripheral initialization error
     2 = System initialization error
     3 = Self test result failed
     4 = Register map value out of range
     5 = Register map address out of range
     6 = Register map write error
     7 = BNO low power mode not available for selected operat ion mode
     8 = Accelerometer power mode not available
     9 = Fusion algorithm configuration error
     A = Sensor configuration error
   */

  if (system_error != 0)
    *system_error = read8(BNO055_SYS_ERR_ADDR);

  delay(200);
}

/*!
 *  @brief Gets the chip revision numbers
 *  @param info
 *         revision info
 */
void Adafruit_BNO055::getRevInfo(adafruit_bno055_rev_info_t *info) {
  uint8_t a, b;

  memset(info, 0, sizeof(adafruit_bno055_rev_info_t));

  /* Check the accelerometer revision */
  info->accel_rev = read8(BNO055_ACCEL_REV_ID_ADDR);

  /* Check the magnetometer revision */
  info->mag_rev = read8(BNO055_MAG_REV_ID_ADDR);

  /* Check the gyroscope revision */
  info->gyro_rev = read8(BNO055_GYRO_REV_ID_ADDR);

  /* Check the SW revision */
  info->bl_rev = read8(BNO055_BL_REV_ID_ADDR);

  a = read8(BNO055_SW_REV_ID_LSB_ADDR);
  b = read8(BNO055_SW_REV_ID_MSB_ADDR);
  info->sw_rev = (((uint16_t)b) << 8) | ((uint16_t)a);
}

/*!
 *  @brief Gets current calibration state. Each value should be a uint8_t
 *         pointer and it will be set to 0 if not calibrated and 3 if
 *         fully calibrated.
 *         See section 34.3.54
```

```
 *  @param sys
 *          Current system calibration status, depends on status of all
     sensors,
 * read-only
 *  @param gyro
 *          Current calibration status of Gyroscope, read-only
 *  @param accel
 *          Current calibration status of Accelerometer, read-only
 *  @param mag
 *          Current calibration status of Magnetometer, read-only
 */
void Adafruit_BNO055::getCalibration(uint8_t *sys, uint8_t *gyro,
                                     uint8_t *accel, uint8_t *mag) {
  uint8_t calData = read8(BNO055_CALIB_STAT_ADDR);
  if (sys != NULL) {
    *sys = (calData >> 6) & 0x03;
  }
  if (gyro != NULL) {
    *gyro = (calData >> 4) & 0x03;
  }
  if (accel != NULL) {
    *accel = (calData >> 2) & 0x03;
  }
  if (mag != NULL) {
    *mag = calData & 0x03;
  }
}

/*!
 *  @brief Gets the temperature in degrees celsius
 *  @return temperature in degrees celsius
 */
int8_t Adafruit_BNO055::getTemp() {
  int8_t temp = (int8_t)(read8(BNO055_TEMP_ADDR));
  return temp;
}

/*!
 *  @brief  Gets a vector reading from the specified source
 *  @param  vector_type
 *          possible vector type values
 *          [VECTOR_ACCELEROMETER
 *           VECTOR_MAGNETOMETER
 *           VECTOR_GYROSCOPE
 *           VECTOR_EULER
 *           VECTOR_LINEARACCEL
 *           VECTOR_GRAVITY]
 *  @return vector from specified source
 */
imu::Vector<3> Adafruit_BNO055::getVector(adafruit_vector_type_t
    vector_type) {
```

```
imu::Vector<3> xyz;
uint8_t buffer[6];
memset(buffer, 0, 6);

int16_t x, y, z;
x = y = z = 0;

/* Read vector data (6 bytes) */
readLen((adafruit_bno055_reg_t)vector_type, buffer, 6);

x = ((int16_t)buffer[0]) | (((int16_t)buffer[1]) << 8);
y = ((int16_t)buffer[2]) | (((int16_t)buffer[3]) << 8);
z = ((int16_t)buffer[4]) | (((int16_t)buffer[5]) << 8);

/*!
 * Convert the value to an appropriate range (section 3.6.4)
 * and assign the value to the Vector type
 */
switch (vector_type) {
case VECTOR_MAGNETOMETER:
  /* 1uT = 16 LSB */
  xyz[0] = ((double)x) / 16.0;
  xyz[1] = ((double)y) / 16.0;
  xyz[2] = ((double)z) / 16.0;
  break;
case VECTOR_GYROSCOPE:
  /* 1dps = 16 LSB */
  xyz[0] = ((double)x) / 16.0;
  xyz[1] = ((double)y) / 16.0;
  xyz[2] = ((double)z) / 16.0;
  break;
case VECTOR_EULER:
  /* 1 degree = 16 LSB */
  xyz[0] = ((double)x) / 16.0;
  xyz[1] = ((double)y) / 16.0;
  xyz[2] = ((double)z) / 16.0;
  break;
case VECTOR_ACCELEROMETER:
  /* 1m/s^2 = 100 LSB */
  xyz[0] = ((double)x) / 100.0;
  xyz[1] = ((double)y) / 100.0;
  xyz[2] = ((double)z) / 100.0;
  break;
case VECTOR_LINEARACCEL:
  /* 1m/s^2 = 100 LSB */
  xyz[0] = ((double)x) / 100.0;
  xyz[1] = ((double)y) / 100.0;
  xyz[2] = ((double)z) / 100.0;
  break;
case VECTOR_GRAVITY:
  /* 1m/s^2 = 100 LSB */
```

```
    xyz[0] = ((double)x) / 100.0;
    xyz[1] = ((double)y) / 100.0;
    xyz[2] = ((double)z) / 100.0;
    break;
  }

  return xyz;
}

/*!
 *  @brief  Gets a quaternion reading from the specified source
 *  @return quaternion reading
 */
imu::Quaternion Adafruit_BNO055::getQuat() {
  uint8_t buffer[8];
  memset(buffer, 0, 8);

  int16_t x, y, z, w;
  x = y = z = w = 0;

  /* Read quat data (8 bytes) */
  readLen(BNO055_QUATERNION_DATA_W_LSB_ADDR, buffer, 8);
  w = (((uint16_t)buffer[1]) << 8) | ((uint16_t)buffer[0]);
  x = (((uint16_t)buffer[3]) << 8) | ((uint16_t)buffer[2]);
  y = (((uint16_t)buffer[5]) << 8) | ((uint16_t)buffer[4]);
  z = (((uint16_t)buffer[7]) << 8) | ((uint16_t)buffer[6]);

  /*!
   * Assign to Quaternion
   * See
   *
       http://ae-bst.resource.bosch.com/media/products/dokumente/bno055/BST_BNO055_DS000_1
   * 3.6.5.5 Orientation (Quaternion)
   */
  const double scale = (1.0 / (1 << 14));
  imu::Quaternion quat(scale * w, scale * x, scale * y, scale * z);
  return quat;
}

/*!
 *  @brief  Provides the sensor_t data for this sensor
 *  @param  sensor
 *          Sensor description
 */
void Adafruit_BNO055::getSensor(sensor_t *sensor) {
  /* Clear the sensor_t object */
  memset(sensor, 0, sizeof(sensor_t));

  /* Insert the sensor name in the fixed length char array */
  strncpy(sensor->name, "BNO055", sizeof(sensor->name) - 1);
  sensor->name[sizeof(sensor->name) - 1] = 0;
```

```
  sensor->version = 1;
  sensor->sensor_id = _sensorID;
  sensor->type = SENSOR_TYPE_ORIENTATION;
  sensor->min_delay = 0;
  sensor->max_value = 0.0F;
  sensor->min_value = 0.0F;
  sensor->resolution = 0.01F;
}

/*!
 *  @brief  Reads the sensor and returns the data as a sensors_event_t
 *  @param  event
 *          Event description
 *  @return always returns true
 */
bool Adafruit_BNO055::getEvent(sensors_event_t *event) {
  /* Clear the event */
  memset(event, 0, sizeof(sensors_event_t));

  event->version = sizeof(sensors_event_t);
  event->sensor_id = _sensorID;
  event->type = SENSOR_TYPE_ORIENTATION;
  event->timestamp = millis();

  /* Get a Euler angle sample for orientation */
  imu::Vector<3> euler = getVector(Adafruit_BNO055::VECTOR_EULER);
  event->orientation.x = euler.x();
  event->orientation.y = euler.y();
  event->orientation.z = euler.z();

  return true;
}

/*!
 *  @brief  Reads the sensor and returns the data as a sensors_event_t
 *  @param  event
 *          Event description
 *  @param  vec_type
 *          specify the type of reading
 *  @return always returns true
 */
bool Adafruit_BNO055::getEvent(sensors_event_t *event,
    adafruit_vector_type_t vec_type)
{
  /* Clear the event */
  memset(event, 0, sizeof(sensors_event_t));

  event->version = sizeof(sensors_event_t);
  event->sensor_id = _sensorID;
  event->timestamp = millis();
```

```
//read the data according to vec_type
imu::Vector<3> vec;
if (vec_type == Adafruit_BNO055::VECTOR_LINEARACCEL)
{
  event->type = SENSOR_TYPE_LINEAR_ACCELERATION;
  vec = getVector(Adafruit_BNO055::VECTOR_LINEARACCEL);

  event->acceleration.x = vec.x();
  event->acceleration.y = vec.y();
  event->acceleration.z = vec.z();
}
else if (vec_type == Adafruit_BNO055::VECTOR_ACCELEROMETER)
{
  event->type = SENSOR_TYPE_ACCELEROMETER;
  vec = getVector(Adafruit_BNO055::VECTOR_ACCELEROMETER);

  event->acceleration.x = vec.x();
  event->acceleration.y = vec.y();
  event->acceleration.z = vec.z();
}
else if (vec_type == Adafruit_BNO055::VECTOR_GRAVITY)
{
  event->type = SENSOR_TYPE_ACCELEROMETER;
  vec = getVector(Adafruit_BNO055::VECTOR_GRAVITY);

  event->acceleration.x = vec.x();
  event->acceleration.y = vec.y();
  event->acceleration.z = vec.z();
}
else if (vec_type == Adafruit_BNO055::VECTOR_EULER)
{
  event->type = SENSOR_TYPE_ORIENTATION;
  vec = getVector(Adafruit_BNO055::VECTOR_EULER);

  event->orientation.x = vec.x();
  event->orientation.y = vec.y();
  event->orientation.z = vec.z();
}
else if (vec_type == Adafruit_BNO055::VECTOR_GYROSCOPE)
{
  event->type = SENSOR_TYPE_ROTATION_VECTOR;
  vec = getVector(Adafruit_BNO055::VECTOR_GYROSCOPE);

  event->gyro.x = vec.x();
  event->gyro.y = vec.y();
  event->gyro.z = vec.z();
}
else if (vec_type == Adafruit_BNO055::VECTOR_MAGNETOMETER)
{
  event->type = SENSOR_TYPE_MAGNETIC_FIELD;
  vec = getVector(Adafruit_BNO055::VECTOR_MAGNETOMETER);
```

```
    event->magnetic.x = vec.x();
    event->magnetic.y = vec.y();
    event->magnetic.z = vec.z();
  }


  return true;
}


/*!
 *  @brief  Reads the sensor's offset registers into a byte array
 *  @param  calibData
 *          Calibration offset (buffer size should be 22)
 *  @return true if read is successful
 */
bool Adafruit_BNO055::getSensorOffsets(uint8_t *calibData) {
  if (isFullyCalibrated()) {
    adafruit_bno055_opmode_t lastMode = _mode;
    setMode(OPERATION_MODE_CONFIG);

    readLen(ACCEL_OFFSET_X_LSB_ADDR, calibData,
        NUM_BNO055_OFFSET_REGISTERS);

    setMode(lastMode);
    return true;
  }
  return false;
}

/*!
 *  @brief  Reads the sensor's offset registers into an offset struct
 *  @param  offsets_type
 *          type of offsets
 *  @return true if read is successful
 */
bool Adafruit_BNO055::getSensorOffsets(
    adafruit_bno055_offsets_t &offsets_type) {
  if (isFullyCalibrated()) {
    adafruit_bno055_opmode_t lastMode = _mode;
    setMode(OPERATION_MODE_CONFIG);
    delay(25);

    /* Accel offset range depends on the G-range:
       +/-2g  = +/- 2000 mg
       +/-4g  = +/- 4000 mg
       +/-8g  = +/- 8000 mg
       +/-1g  = +/- 16000 mg */
    offsets_type.accel_offset_x = (read8(ACCEL_OFFSET_X_MSB_ADDR) << 8) |
                                  (read8(ACCEL_OFFSET_X_LSB_ADDR));
```

```
    offsets_type.accel_offset_y = (read8(ACCEL_OFFSET_Y_MSB_ADDR) << 8) |
                                  (read8(ACCEL_OFFSET_Y_LSB_ADDR));
    offsets_type.accel_offset_z = (read8(ACCEL_OFFSET_Z_MSB_ADDR) << 8) |
                                  (read8(ACCEL_OFFSET_Z_LSB_ADDR));

    /* Magnetometer offset range = +/- 6400 LSB where 1uT = 16 LSB */
    offsets_type.mag_offset_x =
        (read8(MAG_OFFSET_X_MSB_ADDR) << 8) |
            (read8(MAG_OFFSET_X_LSB_ADDR));
    offsets_type.mag_offset_y =
        (read8(MAG_OFFSET_Y_MSB_ADDR) << 8) |
            (read8(MAG_OFFSET_Y_LSB_ADDR));
    offsets_type.mag_offset_z =
        (read8(MAG_OFFSET_Z_MSB_ADDR) << 8) |
            (read8(MAG_OFFSET_Z_LSB_ADDR));

    /* Gyro offset range depends on the DPS range:
      2000 dps = +/- 32000 LSB
      1000 dps = +/- 16000 LSB
       500 dps = +/- 8000 LSB
       250 dps = +/- 4000 LSB
       125 dps = +/- 2000 LSB
       ... where 1 DPS = 16 LSB */
    offsets_type.gyro_offset_x =
        (read8(GYRO_OFFSET_X_MSB_ADDR) << 8) |
            (read8(GYRO_OFFSET_X_LSB_ADDR));
    offsets_type.gyro_offset_y =
        (read8(GYRO_OFFSET_Y_MSB_ADDR) << 8) |
            (read8(GYRO_OFFSET_Y_LSB_ADDR));
    offsets_type.gyro_offset_z =
        (read8(GYRO_OFFSET_Z_MSB_ADDR) << 8) |
            (read8(GYRO_OFFSET_Z_LSB_ADDR));

    /* Accelerometer radius = +/- 1000 LSB */
    offsets_type.accel_radius =
        (read8(ACCEL_RADIUS_MSB_ADDR) << 8) |
            (read8(ACCEL_RADIUS_LSB_ADDR));

    /* Magnetometer radius = +/- 960 LSB */
    offsets_type.mag_radius =
        (read8(MAG_RADIUS_MSB_ADDR) << 8) | (read8(MAG_RADIUS_LSB_ADDR));

    setMode(lastMode);
    return true;
  }
  return false;
}

/*!
 *  @brief  Writes an array of calibration values to the sensor's offset
 *  @param  calibData
```

```
 *         calibration data
 */
void Adafruit_BNO055::setSensorOffsets(const uint8_t *calibData) {
  adafruit_bno055_opmode_t lastMode = _mode;
  setMode(OPERATION_MODE_CONFIG);
  delay(25);

  /* Note: Configuration will take place only when user writes to the last
     byte of each config data pair (ex. ACCEL_OFFSET_Z_MSB_ADDR, etc.).
     Therefore the last byte must be written whenever the user wants to
     changes the configuration. */

  /* A writeLen() would make this much cleaner */
  write8(ACCEL_OFFSET_X_LSB_ADDR, calibData[0]);
  write8(ACCEL_OFFSET_X_MSB_ADDR, calibData[1]);
  write8(ACCEL_OFFSET_Y_LSB_ADDR, calibData[2]);
  write8(ACCEL_OFFSET_Y_MSB_ADDR, calibData[3]);
  write8(ACCEL_OFFSET_Z_LSB_ADDR, calibData[4]);
  write8(ACCEL_OFFSET_Z_MSB_ADDR, calibData[5]);

  write8(MAG_OFFSET_X_LSB_ADDR, calibData[6]);
  write8(MAG_OFFSET_X_MSB_ADDR, calibData[7]);
  write8(MAG_OFFSET_Y_LSB_ADDR, calibData[8]);
  write8(MAG_OFFSET_Y_MSB_ADDR, calibData[9]);
  write8(MAG_OFFSET_Z_LSB_ADDR, calibData[10]);
  write8(MAG_OFFSET_Z_MSB_ADDR, calibData[11]);

  write8(GYRO_OFFSET_X_LSB_ADDR, calibData[12]);
  write8(GYRO_OFFSET_X_MSB_ADDR, calibData[13]);
  write8(GYRO_OFFSET_Y_LSB_ADDR, calibData[14]);
  write8(GYRO_OFFSET_Y_MSB_ADDR, calibData[15]);
  write8(GYRO_OFFSET_Z_LSB_ADDR, calibData[16]);
  write8(GYRO_OFFSET_Z_MSB_ADDR, calibData[17]);

  write8(ACCEL_RADIUS_LSB_ADDR, calibData[18]);
  write8(ACCEL_RADIUS_MSB_ADDR, calibData[19]);

  write8(MAG_RADIUS_LSB_ADDR, calibData[20]);
  write8(MAG_RADIUS_MSB_ADDR, calibData[21]);

  setMode(lastMode);
}

/*!
 *  @brief Writes to the sensor's offset registers from an offset struct
 *  @param offsets_type
 *         accel_offset_x = acceleration offset x
 *         accel_offset_y = acceleration offset y
 *         accel_offset_z = acceleration offset z
 *
 *         mag_offset_x  = magnetometer offset x
```

```
 *          mag_offset_y  = magnetometer offset y
 *          mag_offset_z  = magnetometer offset z
 *
 *          gyro_offset_x = gyroscrope offset x
 *          gyro_offset_y = gyroscrope offset y
 *          gyro_offset_z = gyroscrope offset z
 */
void Adafruit_BNO055::setSensorOffsets(
    const adafruit_bno055_offsets_t &offsets_type) {
  adafruit_bno055_opmode_t lastMode = _mode;
  setMode(OPERATION_MODE_CONFIG);
  delay(25);

  /* Note: Configuration will take place only when user writes to the last
     byte of each config data pair (ex. ACCEL_OFFSET_Z_MSB_ADDR, etc.).
     Therefore the last byte must be written whenever the user wants to
     changes the configuration. */

  write8(ACCEL_OFFSET_X_LSB_ADDR, (offsets_type.accel_offset_x) & 0x0FF);
  write8(ACCEL_OFFSET_X_MSB_ADDR, (offsets_type.accel_offset_x >> 8) &
      0x0FF);
  write8(ACCEL_OFFSET_Y_LSB_ADDR, (offsets_type.accel_offset_y) & 0x0FF);
  write8(ACCEL_OFFSET_Y_MSB_ADDR, (offsets_type.accel_offset_y >> 8) &
      0x0FF);
  write8(ACCEL_OFFSET_Z_LSB_ADDR, (offsets_type.accel_offset_z) & 0x0FF);
  write8(ACCEL_OFFSET_Z_MSB_ADDR, (offsets_type.accel_offset_z >> 8) &
      0x0FF);

  write8(MAG_OFFSET_X_LSB_ADDR, (offsets_type.mag_offset_x) & 0x0FF);
  write8(MAG_OFFSET_X_MSB_ADDR, (offsets_type.mag_offset_x >> 8) & 0x0FF);
  write8(MAG_OFFSET_Y_LSB_ADDR, (offsets_type.mag_offset_y) & 0x0FF);
  write8(MAG_OFFSET_Y_MSB_ADDR, (offsets_type.mag_offset_y >> 8) & 0x0FF);
  write8(MAG_OFFSET_Z_LSB_ADDR, (offsets_type.mag_offset_z) & 0x0FF);
  write8(MAG_OFFSET_Z_MSB_ADDR, (offsets_type.mag_offset_z >> 8) & 0x0FF);

  write8(GYRO_OFFSET_X_LSB_ADDR, (offsets_type.gyro_offset_x) & 0x0FF);
  write8(GYRO_OFFSET_X_MSB_ADDR, (offsets_type.gyro_offset_x >> 8) &
      0x0FF);
  write8(GYRO_OFFSET_Y_LSB_ADDR, (offsets_type.gyro_offset_y) & 0x0FF);
  write8(GYRO_OFFSET_Y_MSB_ADDR, (offsets_type.gyro_offset_y >> 8) &
      0x0FF);
  write8(GYRO_OFFSET_Z_LSB_ADDR, (offsets_type.gyro_offset_z) & 0x0FF);
  write8(GYRO_OFFSET_Z_MSB_ADDR, (offsets_type.gyro_offset_z >> 8) &
      0x0FF);

  write8(ACCEL_RADIUS_LSB_ADDR, (offsets_type.accel_radius) & 0x0FF);
  write8(ACCEL_RADIUS_MSB_ADDR, (offsets_type.accel_radius >> 8) & 0x0FF);

  write8(MAG_RADIUS_LSB_ADDR, (offsets_type.mag_radius) & 0x0FF);
  write8(MAG_RADIUS_MSB_ADDR, (offsets_type.mag_radius >> 8) & 0x0FF);
```

```
  setMode(lastMode);
}

/*!
 *  @brief  Checks of all cal status values are set to 3 (fully calibrated)
 *  @return status of calibration
 */
bool Adafruit_BNO055::isFullyCalibrated() {
  uint8_t system, gyro, accel, mag;
  getCalibration(&system, &gyro, &accel, &mag);

  switch (_mode) {
  case OPERATION_MODE_ACCONLY:
    return (accel == 3);
  case OPERATION_MODE_MAGONLY:
    return (mag == 3);
  case OPERATION_MODE_GYRONLY:
  case OPERATION_MODE_M4G: /* No magnetometer calibration required. */
    return (gyro == 3);
  case OPERATION_MODE_ACCMAG:
  case OPERATION_MODE_COMPASS:
    return (accel == 3 && mag == 3);
  case OPERATION_MODE_ACCGYRO:
  case OPERATION_MODE_IMUPLUS:
    return (accel == 3 && gyro == 3);
  case OPERATION_MODE_MAGGYRO:
    return (mag == 3 && gyro == 3);
  default:
    return (system == 3 && gyro == 3 && accel == 3 && mag == 3);
  }
}

/*!
 *  @brief  Enter Suspend mode (i.e., sleep)
 */
void Adafruit_BNO055::enterSuspendMode() {
  adafruit_bno055_opmode_t modeback = _mode;

  /* Switch to config mode (just in case since this is the default) */
  setMode(OPERATION_MODE_CONFIG);
  delay(25);
  write8(BNO055_PWR_MODE_ADDR, 0x02);
  /* Set the requested operating mode (see section 3.3) */
  setMode(modeback);
  delay(20);
}

/*!
 *  @brief  Enter Normal mode (i.e., wake)
 */
void Adafruit_BNO055::enterNormalMode() {
```

```
  adafruit_bno055_opmode_t modeback = _mode;

  /* Switch to config mode (just in case since this is the default) */
  setMode(OPERATION_MODE_CONFIG);
  delay(25);
  write8(BNO055_PWR_MODE_ADDR, 0x00);
  /* Set the requested operating mode (see section 3.3) */
  setMode(modeback);
  delay(20);
}

/*!
 *  @brief  Writes an 8 bit value over I2C
 */
bool Adafruit_BNO055::write8(adafruit_bno055_reg_t reg, byte value) {
  _wire->beginTransmission(_address);
#if ARDUINO >= 100
  _wire->write((uint8_t)reg);
  _wire->write((uint8_t)value);
#else
  _wire->send(reg);
  _wire->send(value);
#endif
  _wire->endTransmission();

  /* ToDo: Check for error! */
  return true;
}

/*!
 *  @brief  Reads an 8 bit value over I2C
 */
byte Adafruit_BNO055::read8(adafruit_bno055_reg_t reg) {
  byte value = 0;
  Serial.println("Reading");
  _wire->beginTransmission(_address);
  Serial.println("After transmission");
#if ARDUINO >= 100
  Serial.println("Before write reg");
  _wire->write((uint8_t)reg);
  Serial.println("After write reg");
#else
Serial.println("Before send reg");
  _wire->send(reg);
  Serial.println("After send reg");
#endif
  Serial.println("end transmission");
  _wire->endTransmission();
  Serial.println("transmission ended");
  _wire->requestFrom(_address, (byte)1);
#if ARDUINO >= 100
```

```
  value = _wire->read();
#else
  value = _wire->receive();
#endif
 Serial.println("After all read8");

  return value;
}

/*!
 *  @brief  Reads the specified number of bytes over I2C
 */
bool Adafruit_BNO055::readLen(adafruit_bno055_reg_t reg, byte *buffer,
                              uint8_t len) {
  _wire->beginTransmission(_address);
#if ARDUINO >= 100
  _wire->write((uint8_t)reg);
#else
  _wire->send(reg);
#endif
  _wire->endTransmission();
  _wire->requestFrom(_address, (byte)len);

  for (uint8_t i = 0; i < len; i++) {
#if ARDUINO >= 100
    buffer[i] = _wire->read();
#else
    buffer[i] = _wire->receive();
#endif
  }

  /* ToDo: Check for errors! */
  return true;
}




/*!
 *  @file Adafruit_BNO055.h
 *
 *  This is a library for the BNO055 orientation sensor
 *
 *  Designed specifically to work with the Adafruit BNO055 Breakout.
 *
 *  Pick one up today in the adafruit shop!
 *  ------> https://www.adafruit.com/product/2472
 *
 *  These sensors use I2C to communicate, 2 pins are required to
    interface.
 *
```

```
 *  Adafruit invests time and resources providing this open source code,
 *  please support Adafruit andopen-source hardware by purchasing products
 *  from Adafruit!
 *
 *  K.Townsend (Adafruit Industries)
 *
 *  MIT license, all text above must be included in any redistribution
 */

#ifndef __ADAFRUIT_BNO055_H__
#define __ADAFRUIT_BNO055_H__

#include "Arduino.h"
#include <Wire.h>

#include "Adafruit_Sensor.h"
#include "utility/imumaths.h"

/** BNO055 Address A **/
#define BNO055_ADDRESS_A (0x28)
/** BNO055 Address B **/
#define BNO055_ADDRESS_B (0x29)
/** BNO055 ID **/
#define BNO055_ID (0xA0)

/** Offsets registers **/
#define NUM_BNO055_OFFSET_REGISTERS (22)

/** A structure to represent offsets **/
typedef struct {
  int16_t accel_offset_x; /**< x acceleration offset */
  int16_t accel_offset_y; /**< y acceleration offset */
  int16_t accel_offset_z; /**< z acceleration offset */

  int16_t mag_offset_x; /**< x magnetometer offset */
  int16_t mag_offset_y; /**< y magnetometer offset */
  int16_t mag_offset_z; /**< z magnetometer offset */

  int16_t gyro_offset_x; /**< x gyroscrope offset */
  int16_t gyro_offset_y; /**< y gyroscrope offset */
  int16_t gyro_offset_z; /**< z gyroscrope offset */

  int16_t accel_radius; /**< acceleration radius */

  int16_t mag_radius; /**< magnetometer radius */
} adafruit_bno055_offsets_t;

/*!
 *  @brief  Class that stores state and functions for interacting with
 *          BNO055 Sensor
 */
```

```
class Adafruit_BNO055 : public Adafruit_Sensor {
public:
  /** BNO055 Registers **/
  typedef enum {
    /* Page id register definition */
    BNO055_PAGE_ID_ADDR = 0X07,

    /* PAGE0 REGISTER DEFINITION START*/
    BNO055_CHIP_ID_ADDR = 0x00,
    BNO055_ACCEL_REV_ID_ADDR = 0x01,
    BNO055_MAG_REV_ID_ADDR = 0x02,
    BNO055_GYRO_REV_ID_ADDR = 0x03,
    BNO055_SW_REV_ID_LSB_ADDR = 0x04,
    BNO055_SW_REV_ID_MSB_ADDR = 0x05,
    BNO055_BL_REV_ID_ADDR = 0X06,

    /* Accel data register */
    BNO055_ACCEL_DATA_X_LSB_ADDR = 0X08,
    BNO055_ACCEL_DATA_X_MSB_ADDR = 0X09,
    BNO055_ACCEL_DATA_Y_LSB_ADDR = 0X0A,
    BNO055_ACCEL_DATA_Y_MSB_ADDR = 0X0B,
    BNO055_ACCEL_DATA_Z_LSB_ADDR = 0X0C,
    BNO055_ACCEL_DATA_Z_MSB_ADDR = 0X0D,

    /* Mag data register */
    BNO055_MAG_DATA_X_LSB_ADDR = 0X0E,
    BNO055_MAG_DATA_X_MSB_ADDR = 0X0F,
    BNO055_MAG_DATA_Y_LSB_ADDR = 0X10,
    BNO055_MAG_DATA_Y_MSB_ADDR = 0X11,
    BNO055_MAG_DATA_Z_LSB_ADDR = 0X12,
    BNO055_MAG_DATA_Z_MSB_ADDR = 0X13,

    /* Gyro data registers */
    BNO055_GYRO_DATA_X_LSB_ADDR = 0X14,
    BNO055_GYRO_DATA_X_MSB_ADDR = 0X15,
    BNO055_GYRO_DATA_Y_LSB_ADDR = 0X16,
    BNO055_GYRO_DATA_Y_MSB_ADDR = 0X17,
    BNO055_GYRO_DATA_Z_LSB_ADDR = 0X18,
    BNO055_GYRO_DATA_Z_MSB_ADDR = 0X19,

    /* Euler data registers */
    BNO055_EULER_H_LSB_ADDR = 0X1A,
    BNO055_EULER_H_MSB_ADDR = 0X1B,
    BNO055_EULER_R_LSB_ADDR = 0X1C,
    BNO055_EULER_R_MSB_ADDR = 0X1D,
    BNO055_EULER_P_LSB_ADDR = 0X1E,
    BNO055_EULER_P_MSB_ADDR = 0X1F,

    /* Quaternion data registers */
    BNO055_QUATERNION_DATA_W_LSB_ADDR = 0X20,
    BNO055_QUATERNION_DATA_W_MSB_ADDR = 0X21,
```

```
BNO055_QUATERNION_DATA_X_LSB_ADDR = 0X22,
BNO055_QUATERNION_DATA_X_MSB_ADDR = 0X23,
BNO055_QUATERNION_DATA_Y_LSB_ADDR = 0X24,
BNO055_QUATERNION_DATA_Y_MSB_ADDR = 0X25,
BNO055_QUATERNION_DATA_Z_LSB_ADDR = 0X26,
BNO055_QUATERNION_DATA_Z_MSB_ADDR = 0X27,

/* Linear acceleration data registers */
BNO055_LINEAR_ACCEL_DATA_X_LSB_ADDR = 0X28,
BNO055_LINEAR_ACCEL_DATA_X_MSB_ADDR = 0X29,
BNO055_LINEAR_ACCEL_DATA_Y_LSB_ADDR = 0X2A,
BNO055_LINEAR_ACCEL_DATA_Y_MSB_ADDR = 0X2B,
BNO055_LINEAR_ACCEL_DATA_Z_LSB_ADDR = 0X2C,
BNO055_LINEAR_ACCEL_DATA_Z_MSB_ADDR = 0X2D,

/* Gravity data registers */
BNO055_GRAVITY_DATA_X_LSB_ADDR = 0X2E,
BNO055_GRAVITY_DATA_X_MSB_ADDR = 0X2F,
BNO055_GRAVITY_DATA_Y_LSB_ADDR = 0X30,
BNO055_GRAVITY_DATA_Y_MSB_ADDR = 0X31,
BNO055_GRAVITY_DATA_Z_LSB_ADDR = 0X32,
BNO055_GRAVITY_DATA_Z_MSB_ADDR = 0X33,

/* Temperature data register */
BNO055_TEMP_ADDR = 0X34,

/* Status registers */
BNO055_CALIB_STAT_ADDR = 0X35,
BNO055_SELFTEST_RESULT_ADDR = 0X36,
BNO055_INTR_STAT_ADDR = 0X37,

BNO055_SYS_CLK_STAT_ADDR = 0X38,
BNO055_SYS_STAT_ADDR = 0X39,
BNO055_SYS_ERR_ADDR = 0X3A,

/* Unit selection register */
BNO055_UNIT_SEL_ADDR = 0X3B,
BNO055_DATA_SELECT_ADDR = 0X3C,

/* Mode registers */
BNO055_OPR_MODE_ADDR = 0X3D,
BNO055_PWR_MODE_ADDR = 0X3E,

BNO055_SYS_TRIGGER_ADDR = 0X3F,
BNO055_TEMP_SOURCE_ADDR = 0X40,

/* Axis remap registers */
BNO055_AXIS_MAP_CONFIG_ADDR = 0X41,
BNO055_AXIS_MAP_SIGN_ADDR = 0X42,

/* SIC registers */
```

```
  BNO055_SIC_MATRIX_0_LSB_ADDR = 0X43,
  BNO055_SIC_MATRIX_0_MSB_ADDR = 0X44,
  BNO055_SIC_MATRIX_1_LSB_ADDR = 0X45,
  BNO055_SIC_MATRIX_1_MSB_ADDR = 0X46,
  BNO055_SIC_MATRIX_2_LSB_ADDR = 0X47,
  BNO055_SIC_MATRIX_2_MSB_ADDR = 0X48,
  BNO055_SIC_MATRIX_3_LSB_ADDR = 0X49,
  BNO055_SIC_MATRIX_3_MSB_ADDR = 0X4A,
  BNO055_SIC_MATRIX_4_LSB_ADDR = 0X4B,
  BNO055_SIC_MATRIX_4_MSB_ADDR = 0X4C,
  BNO055_SIC_MATRIX_5_LSB_ADDR = 0X4D,
  BNO055_SIC_MATRIX_5_MSB_ADDR = 0X4E,
  BNO055_SIC_MATRIX_6_LSB_ADDR = 0X4F,
  BNO055_SIC_MATRIX_6_MSB_ADDR = 0X50,
  BNO055_SIC_MATRIX_7_LSB_ADDR = 0X51,
  BNO055_SIC_MATRIX_7_MSB_ADDR = 0X52,
  BNO055_SIC_MATRIX_8_LSB_ADDR = 0X53,
  BNO055_SIC_MATRIX_8_MSB_ADDR = 0X54,

  /* Accelerometer Offset registers */
  ACCEL_OFFSET_X_LSB_ADDR = 0X55,
  ACCEL_OFFSET_X_MSB_ADDR = 0X56,
  ACCEL_OFFSET_Y_LSB_ADDR = 0X57,
  ACCEL_OFFSET_Y_MSB_ADDR = 0X58,
  ACCEL_OFFSET_Z_LSB_ADDR = 0X59,
  ACCEL_OFFSET_Z_MSB_ADDR = 0X5A,

  /* Magnetometer Offset registers */
  MAG_OFFSET_X_LSB_ADDR = 0X5B,
  MAG_OFFSET_X_MSB_ADDR = 0X5C,
  MAG_OFFSET_Y_LSB_ADDR = 0X5D,
  MAG_OFFSET_Y_MSB_ADDR = 0X5E,
  MAG_OFFSET_Z_LSB_ADDR = 0X5F,
  MAG_OFFSET_Z_MSB_ADDR = 0X60,

  /* Gyroscope Offset register s*/
  GYRO_OFFSET_X_LSB_ADDR = 0X61,
  GYRO_OFFSET_X_MSB_ADDR = 0X62,
  GYRO_OFFSET_Y_LSB_ADDR = 0X63,
  GYRO_OFFSET_Y_MSB_ADDR = 0X64,
  GYRO_OFFSET_Z_LSB_ADDR = 0X65,
  GYRO_OFFSET_Z_MSB_ADDR = 0X66,

  /* Radius registers */
  ACCEL_RADIUS_LSB_ADDR = 0X67,
  ACCEL_RADIUS_MSB_ADDR = 0X68,
  MAG_RADIUS_LSB_ADDR = 0X69,
  MAG_RADIUS_MSB_ADDR = 0X6A
} adafruit_bno055_reg_t;

/** BNO055 power settings */
```

```
typedef enum {
  POWER_MODE_NORMAL = 0X00,
  POWER_MODE_LOWPOWER = 0X01,
  POWER_MODE_SUSPEND = 0X02
} adafruit_bno055_powermode_t;

/** Operation mode settings **/
typedef enum {
  OPERATION_MODE_CONFIG = 0X00,
  OPERATION_MODE_ACCONLY = 0X01,
  OPERATION_MODE_MAGONLY = 0X02,
  OPERATION_MODE_GYRONLY = 0X03,
  OPERATION_MODE_ACCMAG = 0X04,
  OPERATION_MODE_ACCGYRO = 0X05,
  OPERATION_MODE_MAGGYRO = 0X06,
  OPERATION_MODE_AMG = 0X07,
  OPERATION_MODE_IMUPLUS = 0X08,
  OPERATION_MODE_COMPASS = 0X09,
  OPERATION_MODE_M4G = 0X0A,
  OPERATION_MODE_NDOF_FMC_OFF = 0X0B,
  OPERATION_MODE_NDOF = 0X0C
} adafruit_bno055_opmode_t;

/** Remap settings **/
typedef enum {
  REMAP_CONFIG_P0 = 0x21,
  REMAP_CONFIG_P1 = 0x24, // default
  REMAP_CONFIG_P2 = 0x24,
  REMAP_CONFIG_P3 = 0x21,
  REMAP_CONFIG_P4 = 0x24,
  REMAP_CONFIG_P5 = 0x21,
  REMAP_CONFIG_P6 = 0x21,
  REMAP_CONFIG_P7 = 0x24
} adafruit_bno055_axis_remap_config_t;

/** Remap Signs **/
typedef enum {
  REMAP_SIGN_P0 = 0x04,
  REMAP_SIGN_P1 = 0x00, // default
  REMAP_SIGN_P2 = 0x06,
  REMAP_SIGN_P3 = 0x02,
  REMAP_SIGN_P4 = 0x03,
  REMAP_SIGN_P5 = 0x01,
  REMAP_SIGN_P6 = 0x07,
  REMAP_SIGN_P7 = 0x05
} adafruit_bno055_axis_remap_sign_t;

/** A structure to represent revisions **/
typedef struct {
  uint8_t accel_rev; /**< acceleration rev */
  uint8_t mag_rev; /**< magnetometer rev */
```

```
    uint8_t gyro_rev; /**< gyroscrope rev */
    uint16_t sw_rev; /**< SW rev */
    uint8_t bl_rev;  /**< bootloader rev */
  } adafruit_bno055_rev_info_t;

  /** Vector Mappings **/
  typedef enum {
    VECTOR_ACCELEROMETER = BNO055_ACCEL_DATA_X_LSB_ADDR,
    VECTOR_MAGNETOMETER = BNO055_MAG_DATA_X_LSB_ADDR,
    VECTOR_GYROSCOPE = BNO055_GYRO_DATA_X_LSB_ADDR,
    VECTOR_EULER = BNO055_EULER_H_LSB_ADDR,
    VECTOR_LINEARACCEL = BNO055_LINEAR_ACCEL_DATA_X_LSB_ADDR,
    VECTOR_GRAVITY = BNO055_GRAVITY_DATA_X_LSB_ADDR
  } adafruit_vector_type_t;

  Adafruit_BNO055(int32_t sensorID = -1, uint8_t address =
      BNO055_ADDRESS_A,
                  TwoWire *theWire = &Wire);

  bool begin(adafruit_bno055_opmode_t mode = OPERATION_MODE_NDOF);
  void setMode(adafruit_bno055_opmode_t mode);
  void setAxisRemap(adafruit_bno055_axis_remap_config_t remapcode);
  void setAxisSign(adafruit_bno055_axis_remap_sign_t remapsign);
  void getRevInfo(adafruit_bno055_rev_info_t *);
  void setExtCrystalUse(boolean usextal);
  void getSystemStatus(uint8_t *system_status, uint8_t *self_test_result,
                       uint8_t *system_error);
  void getCalibration(uint8_t *system, uint8_t *gyro, uint8_t *accel,
                      uint8_t *mag);

  imu::Vector<3> getVector(adafruit_vector_type_t vector_type);
  imu::Quaternion getQuat();
  int8_t getTemp();

  /* Adafruit_Sensor implementation */
  bool getEvent(sensors_event_t *);
  bool getEvent(sensors_event_t *, adafruit_vector_type_t);
  void getSensor(sensor_t *);

  /* Functions to deal with raw calibration data */
  bool getSensorOffsets(uint8_t *calibData);
  bool getSensorOffsets(adafruit_bno055_offsets_t &offsets_type);
  void setSensorOffsets(const uint8_t *calibData);
  void setSensorOffsets(const adafruit_bno055_offsets_t &offsets_type);
  bool isFullyCalibrated();

  /* Power managments functions */
  void enterSuspendMode();
  void enterNormalMode();

private:
```

```
  byte read8(adafruit_bno055_reg_t);
  bool readLen(adafruit_bno055_reg_t, byte *buffer, uint8_t len);
  bool write8(adafruit_bno055_reg_t, byte value);

  uint8_t _address;
  TwoWire *_wire;

  int32_t _sensorID;
  adafruit_bno055_opmode_t _mode;
};

#endif
```

A.3.. Adafruit sensor Library

```
#include "Adafruit_Sensor.h"

/***************************************************************************/
/*!
    @brief Prints sensor information to serial console
*/
/***************************************************************************/
void Adafruit_Sensor::printSensorDetails(void) {
  sensor_t sensor;
  getSensor(&sensor);
  Serial.println(F("------------------------------------"));
  Serial.print(F("Sensor:    "));
  Serial.println(sensor.name);
  Serial.print(F("Type:      "));
  switch ((sensors_type_t)sensor.type) {
  case SENSOR_TYPE_ACCELEROMETER:
    Serial.print(F("Acceleration (m/s2)"));
    break;
  case SENSOR_TYPE_MAGNETIC_FIELD:
    Serial.print(F("Magnetic (uT)"));
    break;
  case SENSOR_TYPE_ORIENTATION:
    Serial.print(F("Orientation (degrees)"));
    break;
  case SENSOR_TYPE_GYROSCOPE:
    Serial.print(F("Gyroscopic (rad/s)"));
    break;
  case SENSOR_TYPE_LIGHT:
    Serial.print(F("Light (lux)"));
    break;
  case SENSOR_TYPE_PRESSURE:
    Serial.print(F("Pressure (hPa)"));
    break;
```

```
  case SENSOR_TYPE_PROXIMITY:
    Serial.print(F("Distance (cm)"));
    break;
  case SENSOR_TYPE_GRAVITY:
    Serial.print(F("Gravity (m/s2)"));
    break;
  case SENSOR_TYPE_LINEAR_ACCELERATION:
    Serial.print(F("Linear Acceleration (m/s2)"));
    break;
  case SENSOR_TYPE_ROTATION_VECTOR:
    Serial.print(F("Rotation vector"));
    break;
  case SENSOR_TYPE_RELATIVE_HUMIDITY:
    Serial.print(F("Relative Humidity (%)"));
    break;
  case SENSOR_TYPE_AMBIENT_TEMPERATURE:
    Serial.print(F("Ambient Temp (C)"));
    break;
  case SENSOR_TYPE_OBJECT_TEMPERATURE:
    Serial.print(F("Object Temp (C)"));
    break;
  case SENSOR_TYPE_VOLTAGE:
    Serial.print(F("Voltage (V)"));
    break;
  case SENSOR_TYPE_CURRENT:
    Serial.print(F("Current (mA)"));
    break;
  case SENSOR_TYPE_COLOR:
    Serial.print(F("Color (RGBA)"));
    break;
  }

  Serial.println();
  Serial.print(F("Driver Ver: "));
  Serial.println(sensor.version);
  Serial.print(F("Unique ID: "));
  Serial.println(sensor.sensor_id);
  Serial.print(F("Min Value: "));
  Serial.println(sensor.min_value);
  Serial.print(F("Max Value: "));
  Serial.println(sensor.max_value);
  Serial.print(F("Resolution: "));
  Serial.println(sensor.resolution);
  Serial.println(F("------------------------------------\n"));
}
```

```
/*
 * Copyright (C) 2008 The Android Open Source Project
```

```
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software<
    /span>
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/* Update by K. Townsend (Adafruit Industries) for lighter typedefs, and
 * extended sensor support to include color, voltage and current */

#ifndef _ADAFRUIT_SENSOR_H
#define _ADAFRUIT_SENSOR_H

#ifndef ARDUINO
#include <stdint.h>
#elif ARDUINO >= 100
#include "Arduino.h"
#include "Print.h"
#else
#include "WProgram.h"
#endif

/* Constants */
#define SENSORS_GRAVITY_EARTH (9.80665F) /**< Earth's gravity in m/s^2 */
#define SENSORS_GRAVITY_MOON (1.6F)  /**< The moon's gravity in m/s^2 */
#define SENSORS_GRAVITY_SUN (275.0F) /**< The sun's gravity in m/s^2 */
#define SENSORS_GRAVITY_STANDARD (SENSORS_GRAVITY_EARTH)
#define SENSORS_MAGFIELD_EARTH_MAX                                    \
  (60.0F) /**< Maximum magnetic field on Earth's surface */
#define SENSORS_MAGFIELD_EARTH_MIN                                    \
  (30.0F) /**< Minimum magnetic field on Earth's surface */
#define SENSORS_PRESSURE_SEALEVELHPA                                  \
  (1013.25F) /**< Average sea level pressure is 1013.25 hPa */
#define SENSORS_DPS_TO_RADS                                           \
  (0.017453293F) /**< Degrees/s to rad/s multiplier                  \
                  */
#define SENSORS_RADS_TO_DPS                                           \
  (57.29577793F) /**< Rad/s to degrees/s multiplier */
#define SENSORS_GAUSS_TO_MICROTESLA                                   \
  (100) /**< Gauss to micro-Tesla multiplier */

/** Sensor types */
```

```
typedef enum {
  SENSOR_TYPE_ACCELEROMETER = (1), /**< Gravity + linear acceleration */
  SENSOR_TYPE_MAGNETIC_FIELD = (2),
  SENSOR_TYPE_ORIENTATION = (3),
  SENSOR_TYPE_GYROSCOPE = (4),
  SENSOR_TYPE_LIGHT = (5),
  SENSOR_TYPE_PRESSURE = (6),
  SENSOR_TYPE_PROXIMITY = (8),
  SENSOR_TYPE_GRAVITY = (9),
  SENSOR_TYPE_LINEAR_ACCELERATION =
      (10), /**< Acceleration not including gravity */
  SENSOR_TYPE_ROTATION_VECTOR = (11),
  SENSOR_TYPE_RELATIVE_HUMIDITY = (12),
  SENSOR_TYPE_AMBIENT_TEMPERATURE = (13),
  SENSOR_TYPE_OBJECT_TEMPERATURE = (14),
  SENSOR_TYPE_VOLTAGE = (15),
  SENSOR_TYPE_CURRENT = (16),
  SENSOR_TYPE_COLOR = (17)
} sensors_type_t;

/** struct sensors_vec_s is used to return a vector in a common format. */
typedef struct {
  union {
    float v[3]; ///< 3D vector elements
    struct {
      float x; ///< X component of vector
      float y; ///< Y component of vector
      float z; ///< Z component of vector
    };        ///< Struct for holding XYZ component
    /* Orientation sensors */
    struct {
      float roll; /**< Rotation around the longitudinal axis (the plane
          body, 'X
                    axis'). Roll is positive and increasing when moving
                    downward. -90 degrees <= roll <= 90 degrees */
      float pitch;  /**< Rotation around the lateral axis (the wing span,
          'Y
                       axis'). Pitch is positive and increasing when moving
                       upwards. -180 degrees <= pitch <= 180 degrees) */
      float heading; /**< Angle between the longitudinal axis (the plane
          body)
                       and magnetic north, measured clockwise when viewing
                           from
                       the top of the device. 0-359 degrees */
    };              ///< Struct for holding roll/pitch/heading
  };                ///< Union that can hold 3D vector array, XYZ
      components or
                    ///< roll/pitch/heading
  int8_t status;    ///< Status byte
  uint8_t reserved[3]; ///< Reserved
} sensors_vec_t;
```

```
/** struct sensors_color_s is used to return color data in a common
    format. */
typedef struct {
  union {
    float c[3]; ///< Raw 3-element data
    /* RGB color space */
    struct {
      float r;  /**< Red component */
      float g;  /**< Green component */
      float b;  /**< Blue component */
    };          ///< RGB data in floating point notation
  };            ///< Union of various ways to describe RGB colorspace
  uint32_t rgba; /**< 24-bit RGBA value */
} sensors_color_t;

/* Sensor event (36 bytes) */
/** struct sensor_event_s is used to provide a single sensor event in a
    common
 * format. */
typedef struct {
  int32_t version; /**< must be sizeof(struct sensors_event_t) */
  int32_t sensor_id; /**< unique sensor identifier */
  int32_t type;    /**< sensor type */
  int32_t reserved0; /**< reserved */
  int32_t timestamp; /**< time is in milliseconds */
  union {
    float data[4];           ///< Raw data
    sensors_vec_t acceleration; /**< acceleration values are in meter per
       second
                                per second (m/s^2) */
    sensors_vec_t
       magnetic; /**< magnetic vector values are in micro-Tesla (uT) */
    sensors_vec_t orientation; /**< orientation values are in degrees */
    sensors_vec_t gyro;     /**< gyroscope values are in rad/s */
    float temperature; /**< temperature is in degrees centigrade
       (Celsius) */
    float distance;  /**< distance in centimeters */
    float light;     /**< light in SI lux units */
    float pressure;  /**< pressure in hectopascal (hPa) */
    float relative_humidity; /**< relative humidity in percent */
    float current;         /**< current in milliamps (mA) */
    float voltage;         /**< voltage in volts (V) */
    sensors_color_t color; /**< color in RGB component values */
  };                       ///< Union for the wide ranges of data we can
     carry
} sensors_event_t;

/* Sensor details (40 bytes) */
/** struct sensor_s is used to describe basic information about a specific
 * sensor. */
```

```
typedef struct {
  char name[12];   /**< sensor name */
  int32_t version; /**< version of the hardware + driver */
  int32_t sensor_id; /**< unique sensor identifier */
  int32_t type;    /**< this sensor's type (ex. SENSOR_TYPE_LIGHT) */
  float max_value; /**< maximum value of this sensor's value in SI units
     */
  float min_value; /**< minimum value of this sensor's value in SI units
     */
  float resolution; /**< smallest difference between two values reported
     by this
                     sensor */
  int32_t min_delay; /**< min delay in microseconds between events. zero
     = not a
                      constant rate */
} sensor_t;

/** @brief Common sensor interface to unify various sensors.
 * Intentionally modeled after sensors.h in the Android API:
 *
     https://github.com/android/platform_hardware_libhardware/blob/master/include/hardware
 */
class Adafruit_Sensor {
public:
  // Constructor(s)
  Adafruit_Sensor() {}
  virtual ~Adafruit_Sensor() {}

  // These must be defined by the subclass

  /*! @brief Whether we should automatically change the range (if
     possible) for
     higher precision
     @param enabled True if we will try to autorange */
  virtual void enableAutoRange(bool enabled) {
    (void)enabled; /* suppress unused warning */
  };

  /*! @brief Get the latest sensor event
     @returns True if able to fetch an event */
  virtual bool getEvent(sensors_event_t *) = 0;
  /*! @brief Get info about the sensor itself */
  virtual void getSensor(sensor_t *) = 0;

  void printSensorDetails(void);

private:
  bool _autoRange;
};

#endif
```

A.4.. PID Library

```
/**********************************************************************************************
 * Arduino PID Library - Version 1.2.1
 * by Brett Beauregard <br3ttb@gmail.com> brettbeauregard.com
 *
 * This Library is licensed under the MIT License
 **********************************************************************************************/

#if ARDUINO >= 100
  #include "Arduino.h"
#else
  #include "WProgram.h"
#endif

#include "PID.h"

/*Constructor
    (...)*********************************************************
 *    The parameters specified here are those for for which we can't set
     up
 *    reliable defaults, so we need to have the user set them.
 ************************************************************************/
PID::PID(double* Input, double* Output, double* Setpoint,
        double Kp, double Ki, double Kd, int POn, int ControllerDirection)
{
    myOutput = Output;
    myInput = Input;
    mySetpoint = Setpoint;
    inAuto = false;

    PID::SetOutputLimits(0, 255);     //default output limit corresponds to
                    //the arduino pwm limits

    SampleTime = 100;          //default Controller Sample Time is 0.1
        seconds

    PID::SetControllerDirection(ControllerDirection);
    PID::SetTunings(Kp, Ki, Kd, POn);

    lastTime = millis()-SampleTime;
}

/*Constructor
    (...)*********************************************************
 *    To allow backwards compatability for v1.1, or for people that just
     want
 *    to use Proportional on Error without explicitly saying so
 ************************************************************************/
```

```
PID::PID(double* Input, double* Output, double* Setpoint,
        double Kp, double Ki, double Kd, int ControllerDirection)
    :PID::PID(Input, Output, Setpoint, Kp, Ki, Kd, P_ON_E,
        ControllerDirection)
{

}


/* Compute()
    **********************************************************************
 *    This, as they say, is where the magic happens. this function
     should be called
 *   every time "void loop()" executes. the function will decide for
     itself whether a new
 *   pid Output needs to be computed. returns true when the output is
     computed,
 *   false when nothing has been done.
 **********************************************************************************/
bool PID::Compute()
{
   if(!inAuto) return false;
   unsigned long now = millis();
   unsigned long timeChange = (now - lastTime);
   if(timeChange>=SampleTime)
   {
      /*Compute all the working error variables*/
      double input = *myInput;
      double error = *mySetpoint - input;
      double dInput = (input - lastInput);
      if(error>180) {error = error-360;}
      if(error<-180) {error = 360+error;}
      outputSum+= (ki * error);
      /*Add Proportional on Measurement, if P_ON_M is specified*/
      if(!pOnE) outputSum-= kp * dInput;

      if(outputSum > outMax) outputSum= outMax;
      else if(outputSum < outMin) outputSum= outMin;

      /*Add Proportional on Error, if P_ON_E is specified*/
     double output;
      if(pOnE) output = kp * error;
      else output = 0;

      /*Compute Rest of PID Output*/
      output += outputSum - kd * dInput;

      if(output > outMax) output = outMax;
      else if(output < outMin) output = outMin;
      *myOutput = output;
```

```
      /*Remember some variables for next time*/
      lastInput = input;
      lastTime = now;
      error_PID = error;
      return true;
   }
   else return false;
}

/*
    SetTunings(...)*************************************************************
 * This function allows the controller's dynamic performance to be
     adjusted.
 * it's called automatically from the constructor, but tunings can also
 * be adjusted on the fly during normal operation
 ******************************************************************************/
void PID::SetTunings(double Kp, double Ki, double Kd, int POn)
{
   if (Kp<0 || Ki<0 || Kd<0) return;

   pOn = POn;
   pOnE = POn == P_ON_E;

   dispKp = Kp; dispKi = Ki; dispKd = Kd;

   double SampleTimeInSec = ((double)SampleTime)/1000;
   kp = Kp;
   ki = Ki * SampleTimeInSec;
   kd = Kd / SampleTimeInSec;

  if(controllerDirection ==REVERSE)
   {
      kp = (0 - kp);
      ki = (0 - ki);
      kd = (0 - kd);
   }
}

/*
    SetTunings(...)*************************************************************
 * Set Tunings using the last-rembered POn setting
 ******************************************************************************/
void PID::SetTunings(double Kp, double Ki, double Kd){
    SetTunings(Kp, Ki, Kd, pOn);
}

/* SetSampleTime(...)
    *********************************************************
 * sets the period, in Milliseconds, at which the calculation is performed
 ******************************************************************************/
void PID::SetSampleTime(int NewSampleTime)
```

```
{
   if (NewSampleTime > 0)
   {
      double ratio = (double)NewSampleTime
                     / (double)SampleTime;
      ki *= ratio;
      kd /= ratio;
      SampleTime = (unsigned long)NewSampleTime;
   }
}

/*
    SetOutputLimits(...)****************************************************
 *    This function will be used far more often than SetInputLimits.
     while
 *  the input to the controller will generally be in the 0-1023 range
     (which is
 *  the default already,) the output will be a little different. maybe
     they'll
 *  be doing a time window and will need 0-8000 or something. or maybe
     they'll
 *  want to clamp it from 0-125. who knows. at any rate, that can all be
     done
 *  here.
 **************************************************************************/
void PID::SetOutputLimits(double Min, double Max)
{
   if(Min >= Max) return;
   outMin = Min;
   outMax = Max;

   if(inAuto)
   {
     if(*myOutput > outMax) *myOutput = outMax;
     else if(*myOutput < outMin) *myOutput = outMin;

     if(outputSum > outMax) outputSum= outMax;
     else if(outputSum < outMin) outputSum= outMin;
   }
}

/*
    SetMode(...)****************************************************************
 * Allows the controller Mode to be set to manual (0) or Automatic
     (non-zero)
 * when the transition from manual to auto occurs, the controller is
 * automatically initialized
 **************************************************************************/
void PID::SetMode(int Mode)
{
    bool newAuto = (Mode == AUTOMATIC);
```

```
    if(newAuto && !inAuto)
    {  /*we just went from manual to auto*/
        PID::Initialize();
    }
    inAuto = newAuto;
}

/*
    Initialize()****************************************************************
 *  does all the things that need to happen to ensure a bumpless transfer
 *  from manual to automatic mode.
 ******************************************************************************/
void PID::Initialize()
{
   outputSum = *myOutput;
   lastInput = *myInput;
   if(outputSum > outMax) outputSum = outMax;
   else if(outputSum < outMin) outputSum = outMin;
}

/*
    SetControllerDirection(...)*************************************************
 * The PID will either be connected to a DIRECT acting process (+Output
    leads
 * to +Input) or a REVERSE acting process(+Output leads to -Input.) we
    need to
 * know which one, because otherwise we may increase the output when we
    should
 * be decreasing. This is called from the constructor.
 ******************************************************************************/
void PID::SetControllerDirection(int Direction)
{
   if(inAuto && Direction !=controllerDirection)
   {
      kp = (0 - kp);
      ki = (0 - ki);
      kd = (0 - kd);
   }
   controllerDirection = Direction;
}

/* Status
    Funcions****************************************************************
 * Just because you set the Kp=-1 doesn't mean it actually happened. these
 * functions query the internal state of the PID. they're here for display
 * purposes. this are the functions the PID Front-end uses for example
 ******************************************************************************/
double PID::GetKp(){ return dispKp; }
double PID::GetKi(){ return dispKi;}
double PID::GetKd(){ return dispKd;}
int PID::GetMode(){ return inAuto ? AUTOMATIC : MANUAL;}
```

```
int PID::GetDirection(){ return controllerDirection;}


#ifndef PID_v1_h
#define PID_v1_h
#define LIBRARY_VERSION 1.2.1

class PID
{


  public:

  //Constants used in some of the functions below
  #define AUTOMATIC 1
  #define MANUAL 0
  #define DIRECT 0
  #define REVERSE 1
  #define P_ON_M 0
  #define P_ON_E 1

  //commonly used functions
     **************************************************************************
    PID(double*, double*, double*,     // * constructor. links the PID to
        the Input, Output, and
        double, double, double, int, int);// Setpoint. Initial tuning
            parameters are also set here.
                                       //   (overload for specifying
                                           proportional mode)

    PID(double*, double*, double*,     // * constructor. links the PID to
        the Input, Output, and
        double, double, double, int);  //   Setpoint. Initial tuning
            parameters are also set here

    void SetMode(int Mode);            // * sets PID to either Manual (0)
        or Auto (non-0)

    bool Compute();                    // * performs the PID calculation.
        it should be
                                       //   called every time loop()
                                           cycles. ON/OFF and
                                       //   calculation frequency can be
                                           set using SetMode
                                       //   SetSampleTime respectively

    void SetOutputLimits(double, double); // * clamps the output to a
        specific range. 0-255 by default, but
                                       //   it's likely the user will want
                                           to change this depending on
                                       //   the application
```

```
//available but not commonly used functions
    *****************************************************
  void SetTunings(double, double,    // * While most users will set the
     tunings once in the
                  double);           //   constructor, this function
                     gives the user the option
                                     //   of changing tunings during
                                        runtime for Adaptive control
  void SetTunings(double, double,    // * overload for specifying
     proportional mode
                  double, int);

void SetControllerDirection(int); // * Sets the Direction, or "Action"
    of the controller. DIRECT
                  //   means the output will increase when error is
                     positive. REVERSE
                  //   means the opposite. it's very unlikely that this
                     will be needed
                  //   once it is set in the constructor.
  void SetSampleTime(int);           // * sets the frequency, in
     Milliseconds, with which
                                     //   the PID calculation is
                                        performed. default is 100



//Display functions
    ****************************************************************
double GetKp();           // These functions query the pid for interal
    values.
double GetKi();           //  they were created mainly for the pid
    front-end,
double GetKd();           // where it's important to know what is
    actually
int GetMode();            //  inside the PID.
int GetDirection();       //
double GetError();
double error_PID;
private:
void Initialize();

double dispKp;      // * we'll hold on to the tuning parameters in
    user-entered
double dispKi;      //   format for display purposes
double dispKd;      //

double kp;                // * (P)roportional Tuning Parameter
  double ki;                // * (I)ntegral Tuning Parameter
```

```
    double kd;                // * (D)erivative Tuning Parameter

  int controllerDirection;
  int pOn;

    double *myInput;            // * Pointers to the Input, Output, and
        Setpoint variables
    double *myOutput;           //   This creates a hard link between the
        variables and the
    double *mySetpoint;         //   PID, freeing the user from having to
        constantly tell us
                                //   what these values are. with pointers
                                   we'll just know.

  unsigned long lastTime;
  double outputSum, lastInput;

  unsigned long SampleTime;
  double outMin, outMax;
  bool inAuto, pOnE;
};
#endif
```

A.5.. API Server

This code downloads the data from an API server and displays it in the terminal. It is programmed in Python

```
from flask import Flask, request
from flask_restful import Resource, Api
from json import dumps

app = Flask(__name__)
api = Api(app)
dict_sensor = {"power":[], "angles":[], "time":[], "error":[]}


@app.route('/data/upload', methods=['POST'])
def create_task():
    print(request.json)
    dict_sensor["power"].append(request.json["power"])
    dict_sensor["angles"].append(request.json["angle"])
    dict_sensor["time"].append(request.json["time"])
    dict_sensor["error"].append(request.json["error"])
    return "hello"
@app.route('/stop', methods=['GET'])
def stop_task():
    sensor_data_file = open("sensor_data_file.csv","w")
```

```
    sensor_data_file.write("Power, Angle, Time, Error\n")
    i = 0
    for item in dict_sensor["power"]:
        #sensor_data_file.write("{},{}".format(item,dict_sensor["angle"][i]))
        sensor_data_file.write(str(item)+ ", "
            +str(dict_sensor["angles"][i])+ ", "
            +str(dict_sensor["time"][i])+ ", "
            +str(dict_sensor["error"][i])+"\n")
        i+=1

    return "hello"

if __name__ == '__main__':
     app.run(host='192.168.1.123',port='5002')
```

A.6.. Plots PID

This codes plot the data downloaded from the API. Code of the combined plots:

```
import pandas as pd
import matplotlib.pyplot as plt
import sys
import re
df_list = []
names=[]
colors=['r','b','y','g']
kp_list=[]
ki_list=[]
kd_list=[]
for files in sys.argv[1:]:
    df_list.append(pd.read_csv(files))
    name = re.sub(".csv","",files)
    names.append(name)
    param = re.sub("sensor.*file_","",name)
    param = re.sub("_[0-9]{4}_[0-9]{2}_[0-9]{2}","",param)
    kp_list.append(re.sub("kp","",re.sub("_ki.*","",param)))
    ki_list.append(re.sub("kp.*ki(.*)_.*",r"\1",param))
    kd_list.append(re.sub("kp.*ki.*kd(.*)",r"\1",param))
if len(set(ki_list)) > 1:
    title="kd={}, kp={}".format(kd_list[0],kp_list[0])
    filename = "kd{}_kp{}".format(kd_list[0],kp_list[0])
    values = ki_list
    mode='ki'
if len(set(kd_list)) > 1:
    title="ki={}, kp={}".format(ki_list[0],kp_list[0])
    filename = "ki{}_kp{}".format(ki_list[0],kp_list[0])
    values = kd_list
```

```
    mode='kd'
if len(set(kp_list)) > 1:
    title="kd={}, ki={}".format(kd_list[0],ki_list[0])
    filename = "kd{}_ki{}".format(kd_list[0],ki_list[0])
    values = kp_list
    mode='kp'
fig = plt.figure()
for color,frame,value in zip(colors,df_list,values):
    frame[' Time'] = frame[' Time']/1000
    frame[' Time'] = frame[' Time'] - frame[' Time'][0]
    plt.plot(frame[' Time'], frame[' Error'],':',color=color)
    plt.plot(frame[' Time'], frame[' Angle'],'-',color=color,
        label="{}={}".format(mode,value))
plt.title(title)
plt.legend()
plt.grid()
plt.xlabel("Time [s]")
plt.ylabel("Angle []")
fig.savefig("angle_{}.eps".format(filename))
fig.savefig("angle_{}.png".format(filename))

fig2 = plt.figure()
for color,frame,value in zip(colors,df_list,values):
    plt.plot(frame[' Time'], frame['Power'],'-',color=color,
        label="{}={}".format(mode,value))
plt.title(title)
plt.legend()
plt.grid()
plt.xlabel("Time [s]")
plt.ylabel(u"$T_{pwm}$ [\u00b5s]")
fig2.savefig("power_{}.eps".format(filename))
fig2.savefig("power_{}.png".format(filename))
plt.show()
```

Plots the graphics for different tuning parameters and 300º angle.

```
import pandas as pd
import matplotlib.pyplot as plt
import sys
import re
df = pd.read_csv(sys.argv[1])
name = re.sub(".csv","",sys.argv[1])
param = re.sub("sensor.*file_","",name)
param = re.sub("_[0-9]{4}_[0-9]{2}_[0-9]{2}","",param)
kp = re.sub("kp","",re.sub("_ki.*","",param))
print(kp)
ki = re.sub("kp.*ki(.*)_.*",r"\1",param)
print(ki)
kd = re.sub("kp.*ki.*kd(.*)",r"\1",param)
print(kd)
df[' Time'] = df[' Time']/1000
```

```
df.plot(x = ' Time', y = [' Error', ' Angle'])
plt.legend(["Error", "Angle"])
plt.xlabel("Time [s]")
plt.ylabel("Angle []")
plt.title("Kp: {}, Ki: {}, Kd: {}".format(kp,ki,kd))
plt.grid()
plt.savefig("angle_kp{}_ki{}_kd{}.eps".format(kp,ki,kd))
df.plot(x= ' Time', y = 'Power')
plt.legend(["Power"])
plt.xlabel("Time [s]")
plt.grid()
plt.title("Kp: {}, Ki: {}, Kd: {}".format(kp,ki,kd))
plt.savefig("power_kp{}_ki{}_kd{}.eps".format(kp,ki,kd))
```

Plots the graphics for different angles

```
import pandas as pd
import matplotlib.pyplot as plt
import sys
import re
df = pd.read_csv(sys.argv[1])
name = re.sub(".csv","",sys.argv[1])
param = re.sub("sensor.*file_","",name)
param = re.sub("_[0-9]{4}_[0-9]{2}_[0-9]{2}","",param)
kp = re.sub("kp","",re.sub("_ki.*","",param))
print(kp)
ki = re.sub("kp.*ki(.*)_.*",r"\1",param)
print(ki)
kd = re.sub("kp.*ki.*kd(.*)",r"\1",param)
print(kd)
df[' Time'] = df[' Time']/1000
df.plot(x = ' Time', y = [' Error', ' Angle'])
plt.legend(["Error", "Angle"])
plt.xlabel("Time [s]")
plt.ylabel("Angle []")
plt.title("Kp: {}, Ki: {}, Kd: {}".format(kp,ki,kd))
plt.grid()
plt.savefig("angle_kp{}_ki{}_kd{}.eps".format(kp,ki,kd))
df.plot(x= ' Time', y = 'Power')
plt.legend(["Power"])
plt.xlabel("Time [s]")
plt.grid()
plt.title("Kp: {}, Ki: {}, Kd: {}".format(kp,ki,kd))
plt.savefig("power_kp{}_ki{}_kd{}.eps".format(kp,ki,kd))
```

APPENDIX B. FIGURES PID PERFORMANCE

In this annex the figures for each tuning parameter can be seen.

B.1.. Variation of K_p

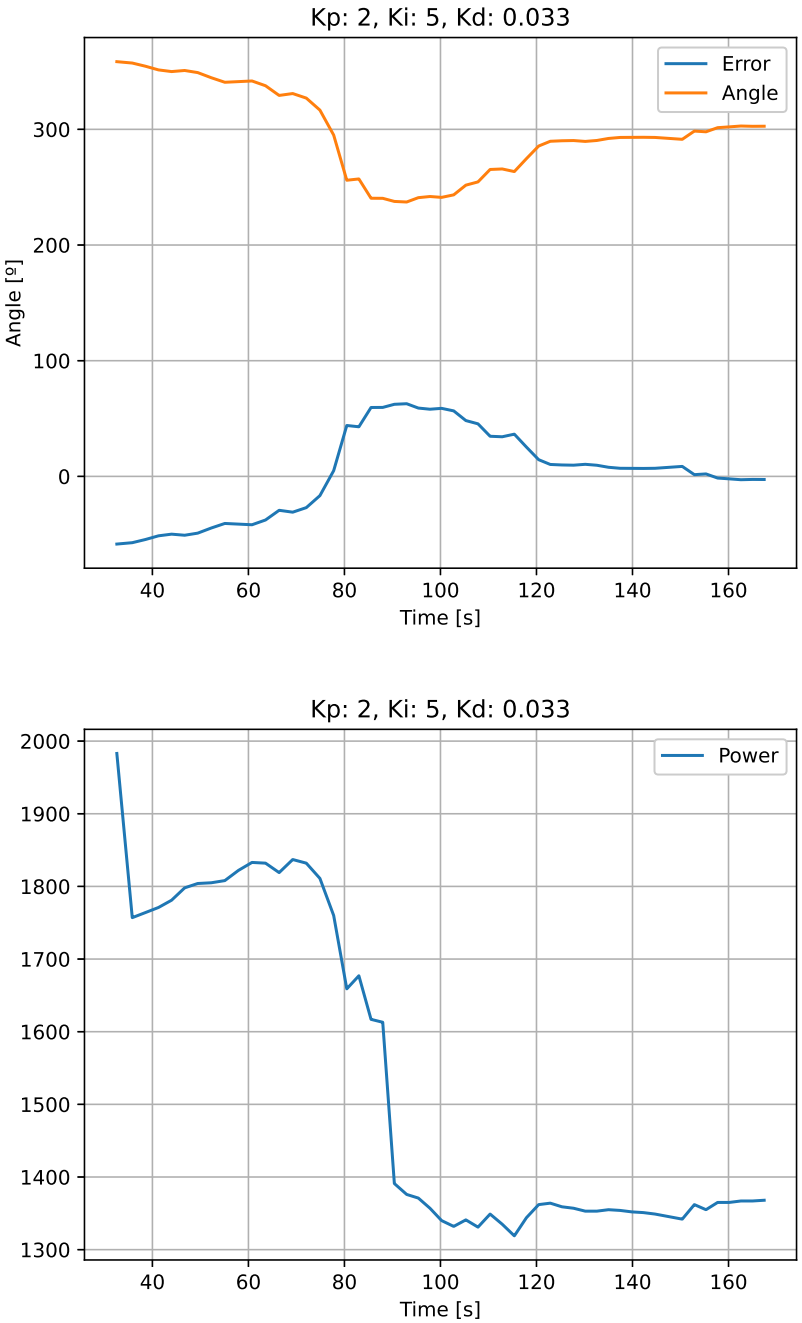


Figure B.1: Sensor data k_p: 2; k_i: 5; k_d: 0.033.

In this case the first value of angle is 358º approximately instead of 0º because there is a delay when receiving the data of the sensor. This may happen in several several tests.

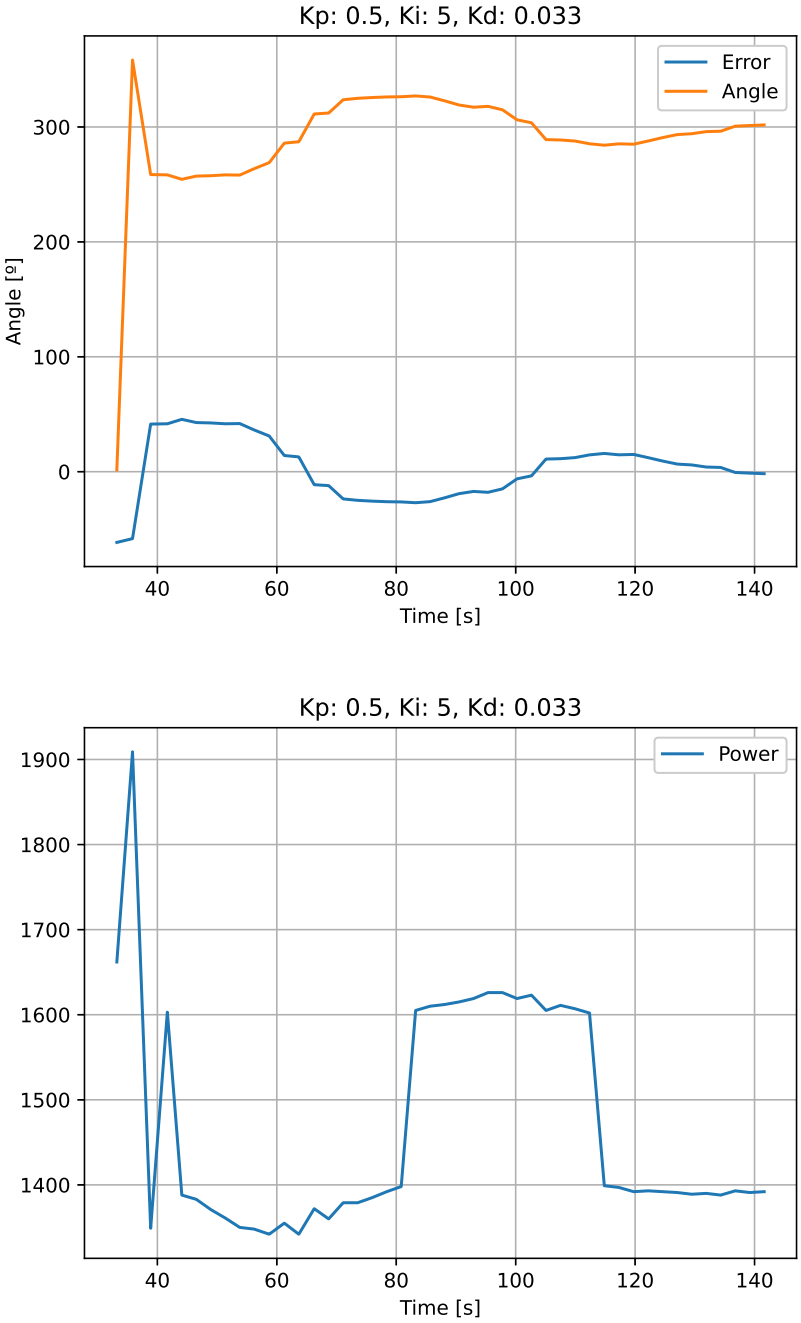
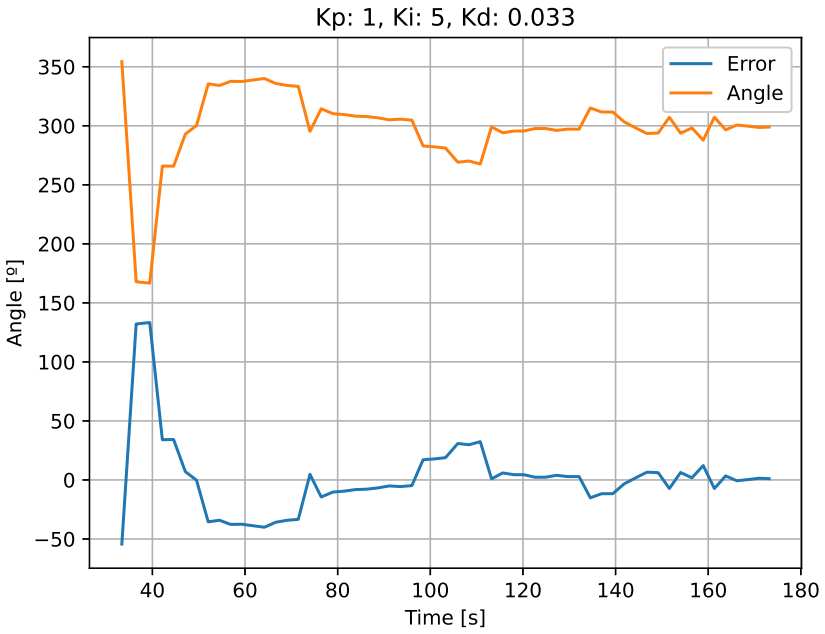


Figure B.2: Sensor data k_p: 0.5; k_i: 5; k_d: 0.033



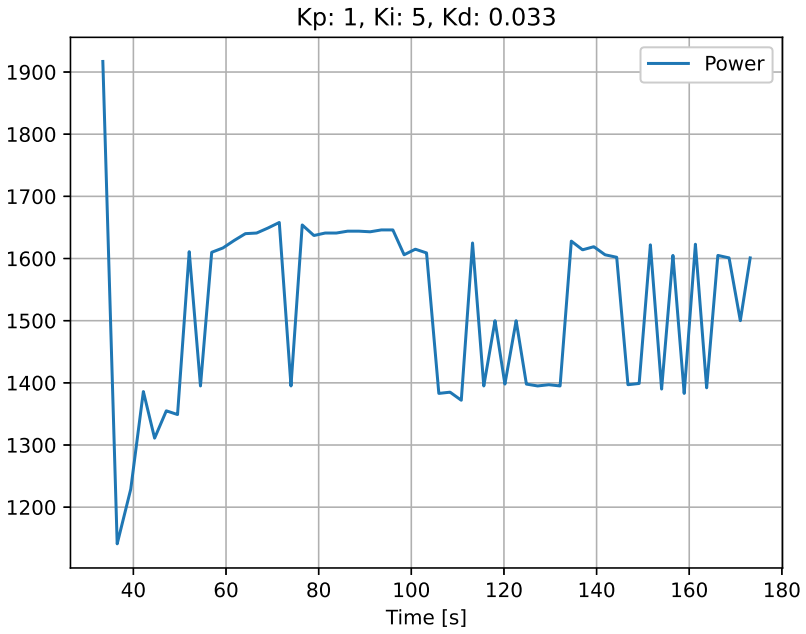


Figure B.3: Sensor data k_p: 1; k_i: 5; k_d: 0.033

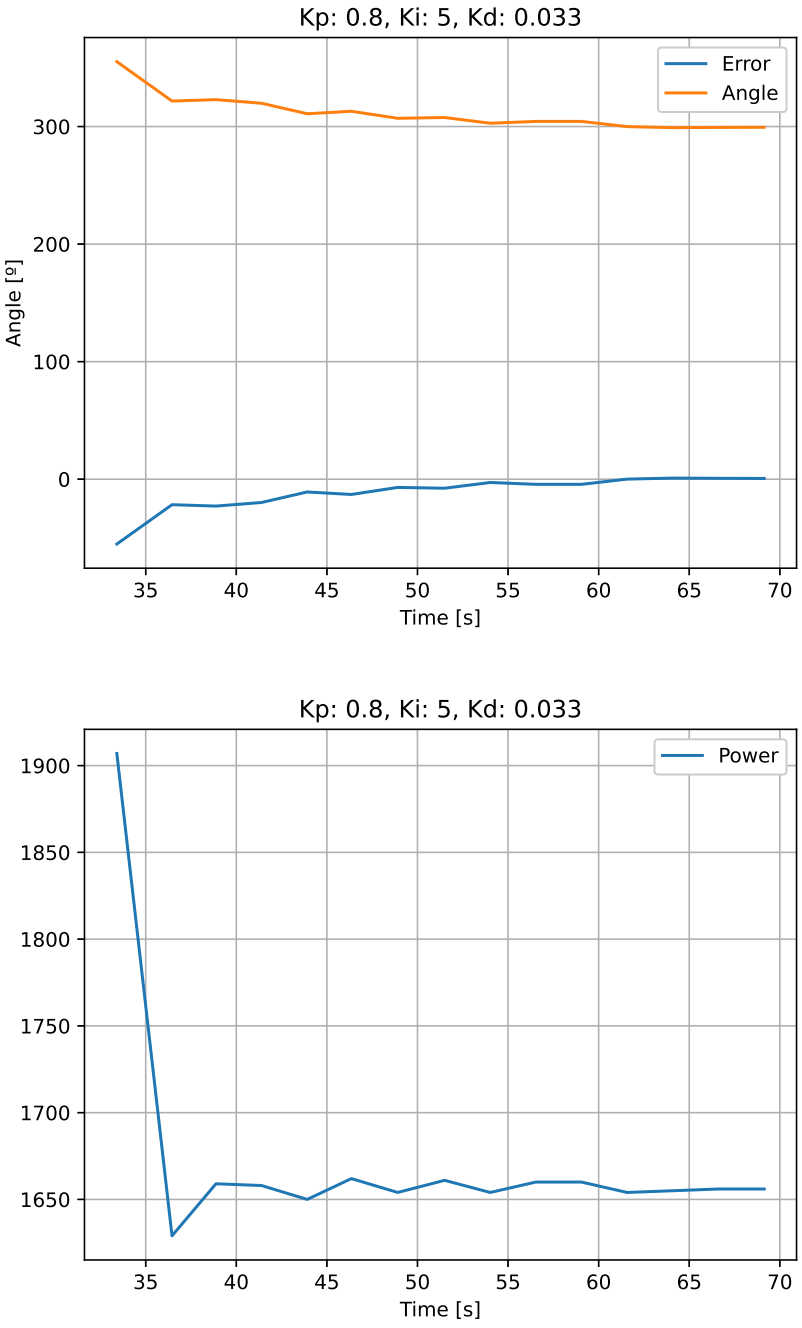


Figure B.4: Sensor data k_p: 0.8; k_i: 5; k_d: 0.033

This is a good value of PID. The setpoint is reached fast and the power consumed by the engine is quite low.

B.2.. Variation of K_i

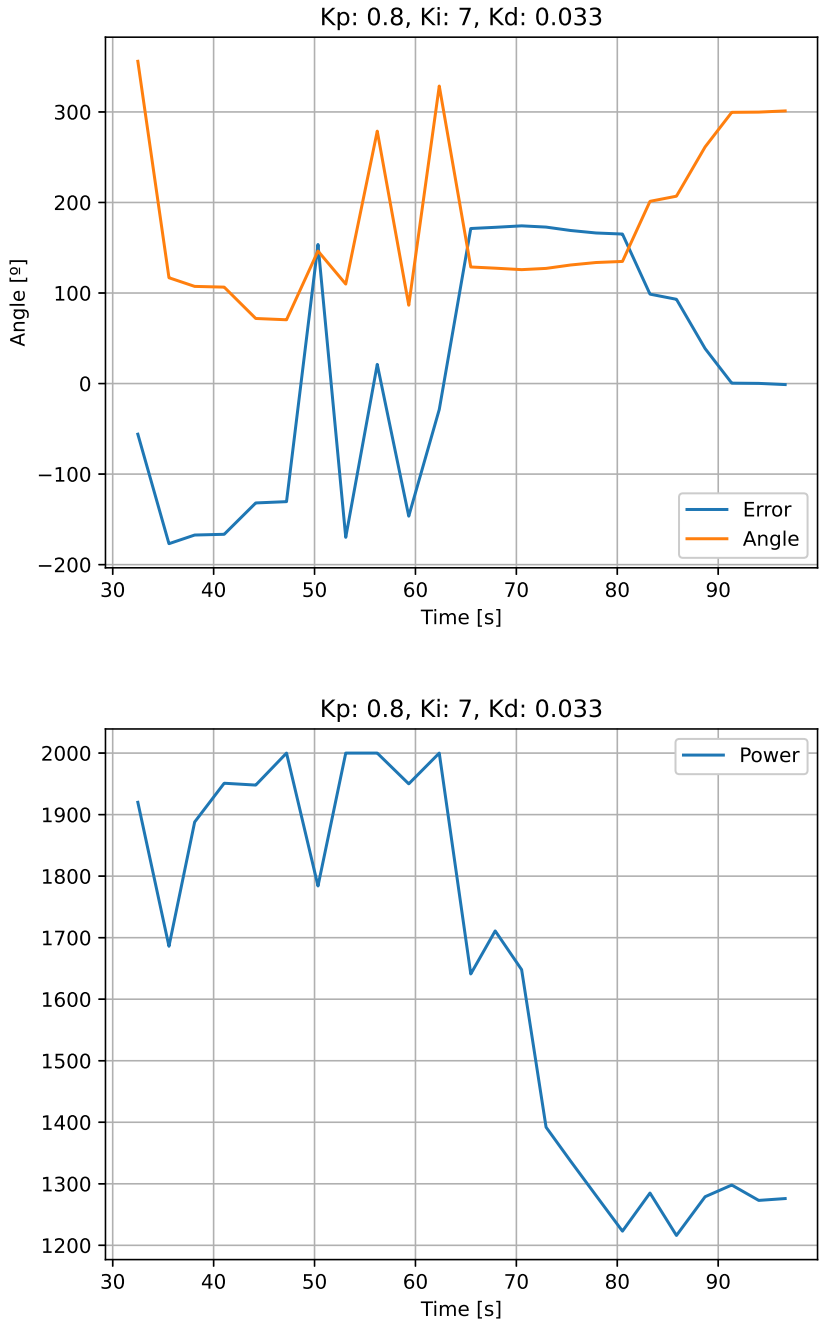
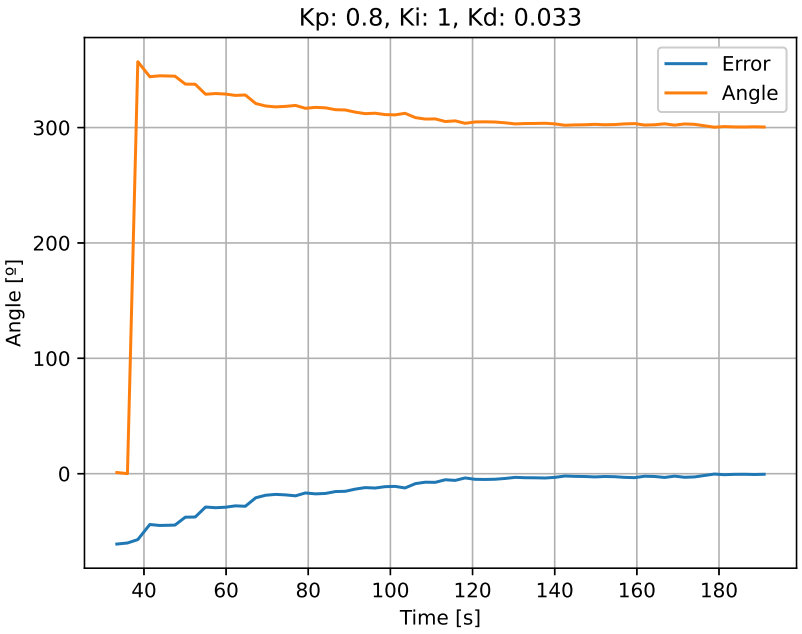


Figure B.5: Sensor data k_p: 0.8; k_i: 7; k_d: 0.033

In this test as the Integrator is too big, the overshoot is high, which translates into a hole turn of the system several times.



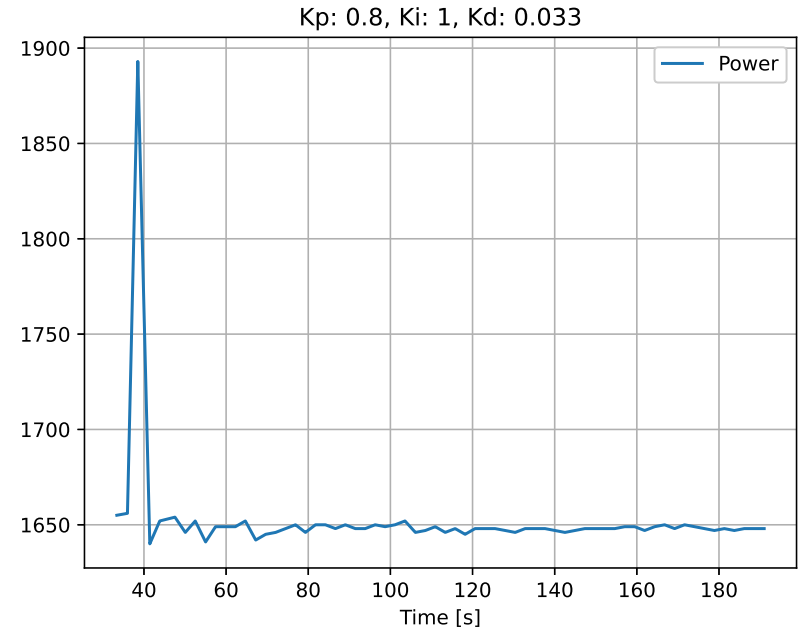
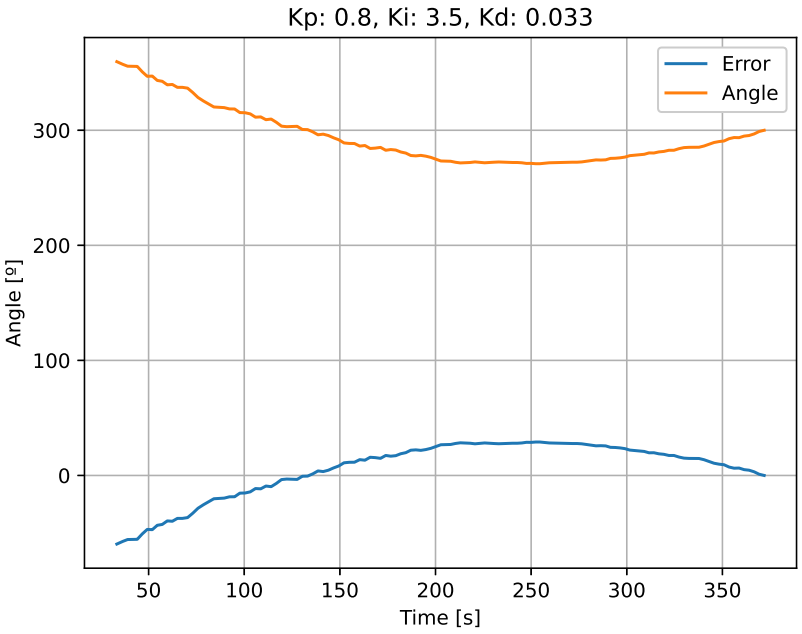


Figure B.6: Sensor data k_p: 0.8; k_i: 1; k_d: 0.033



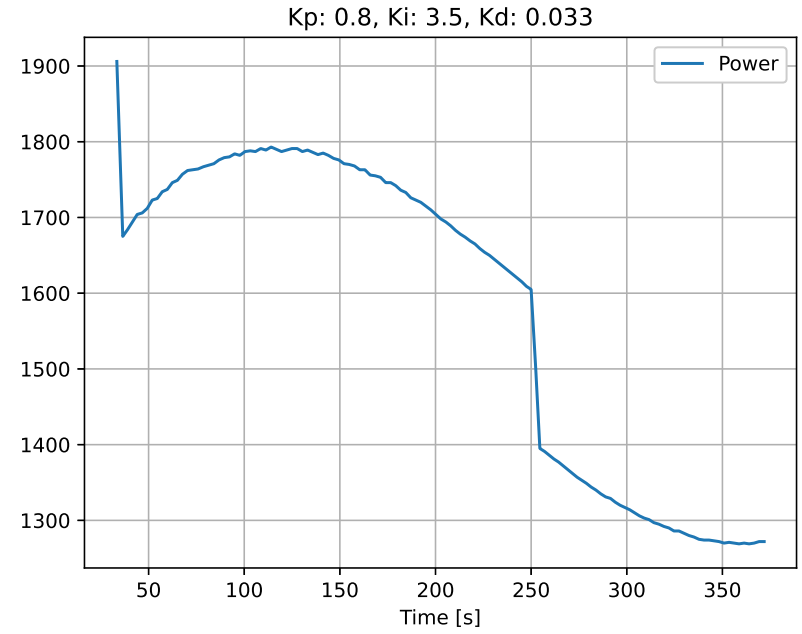
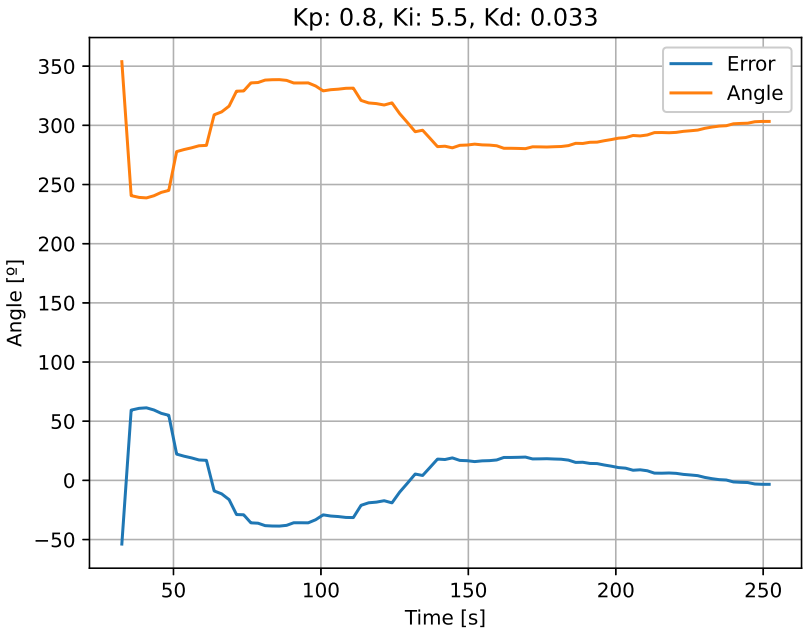


Figure B.7: Sensor data k_p: 0.8; k_i: 3.5; k_d: 0.033



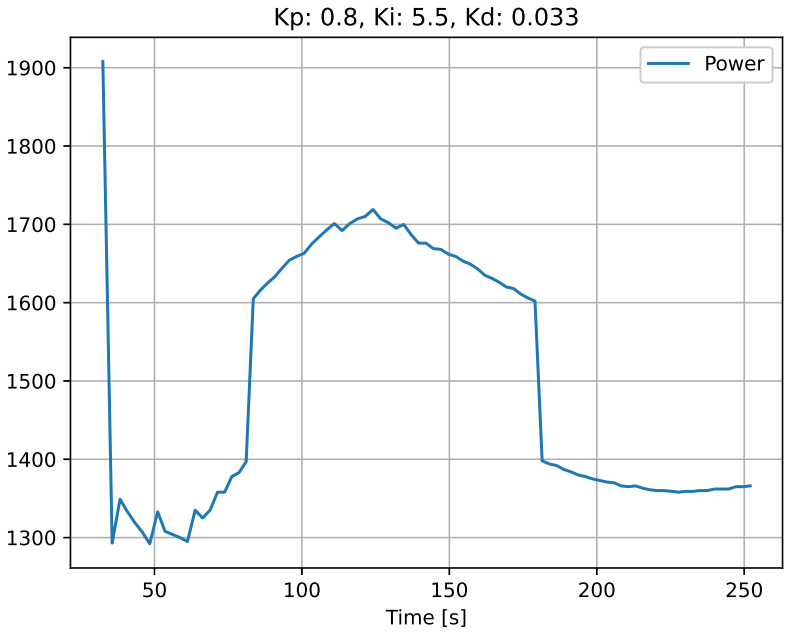


Figure B.8: Sensor data k_p: 0.8; k_i: 5.5; k_d: 0.033

B.3.. Variation K_d

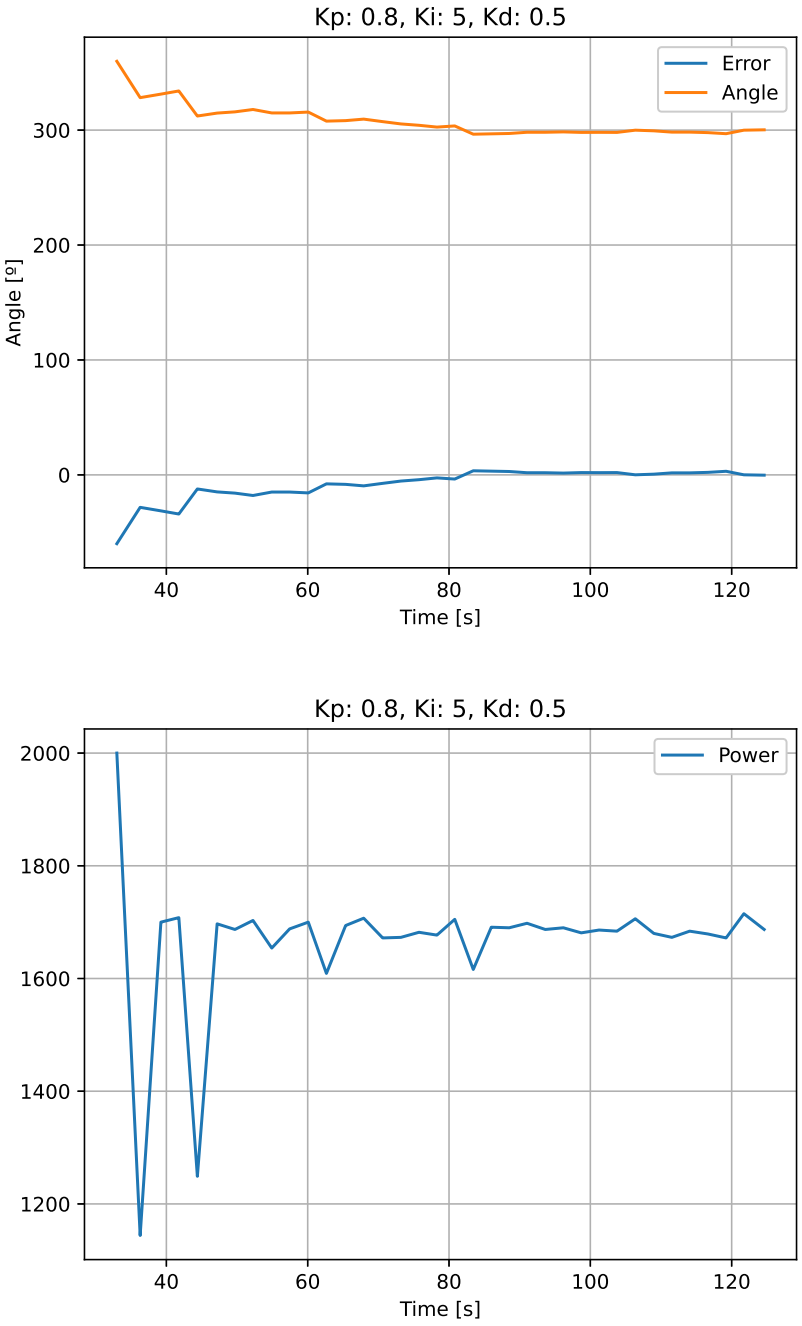
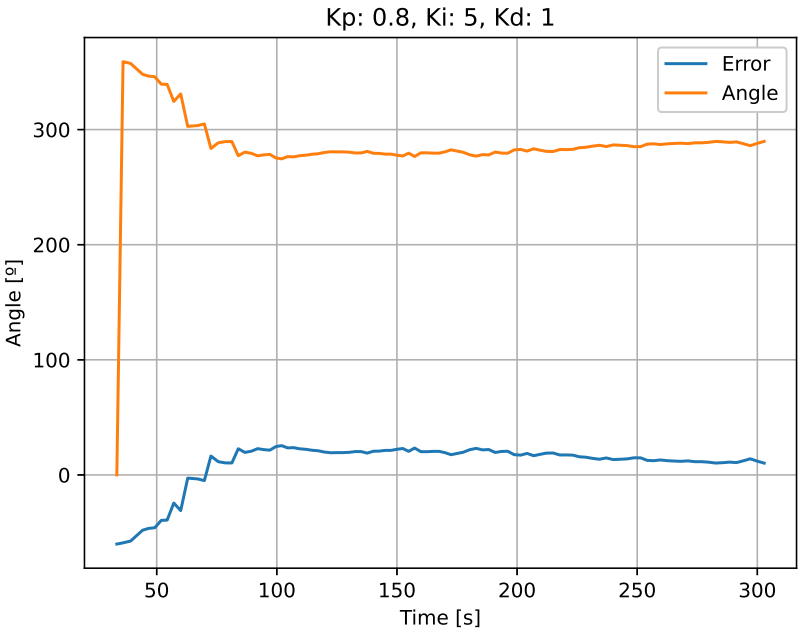


Figure B.9: Sensor data k_p: 0.8; k_i: 5; k_d: 0.5



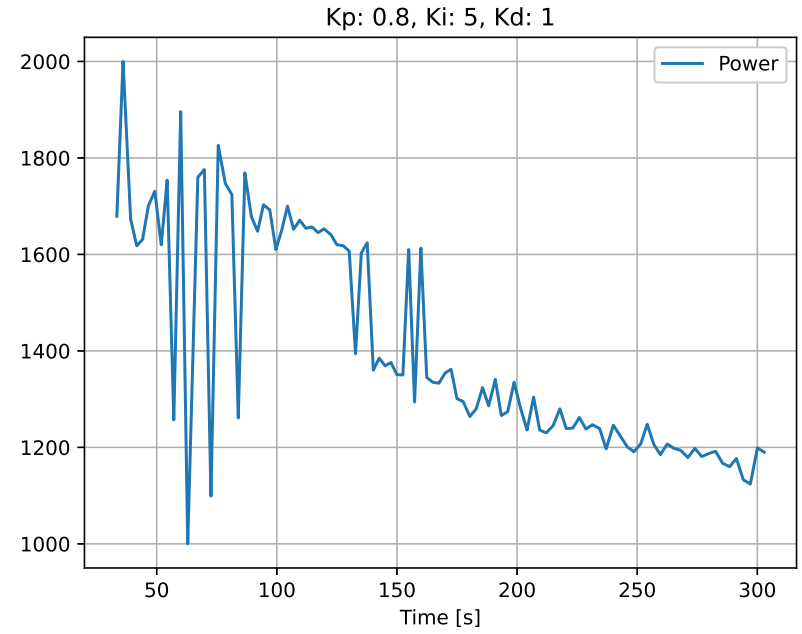
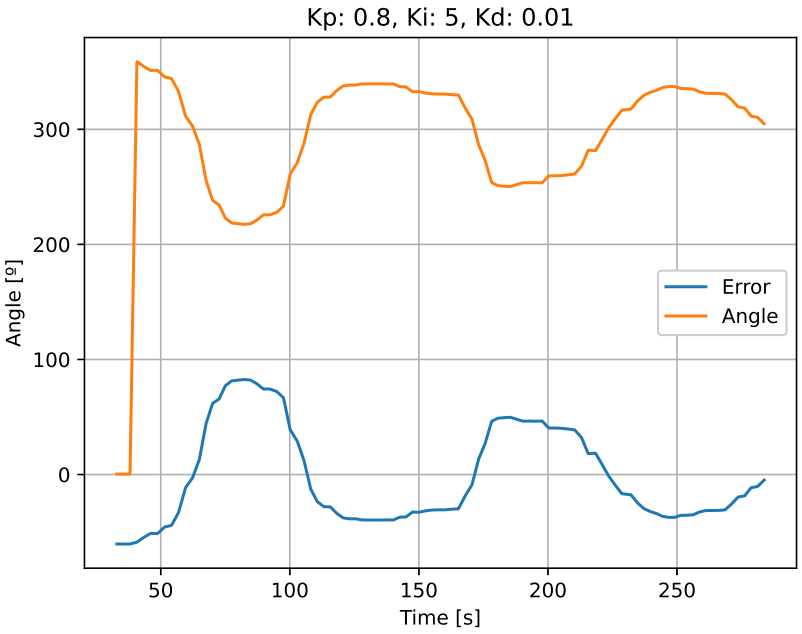


Figure B.10: Sensor data k_p: 0.8; k_i: 5; k_d: 1



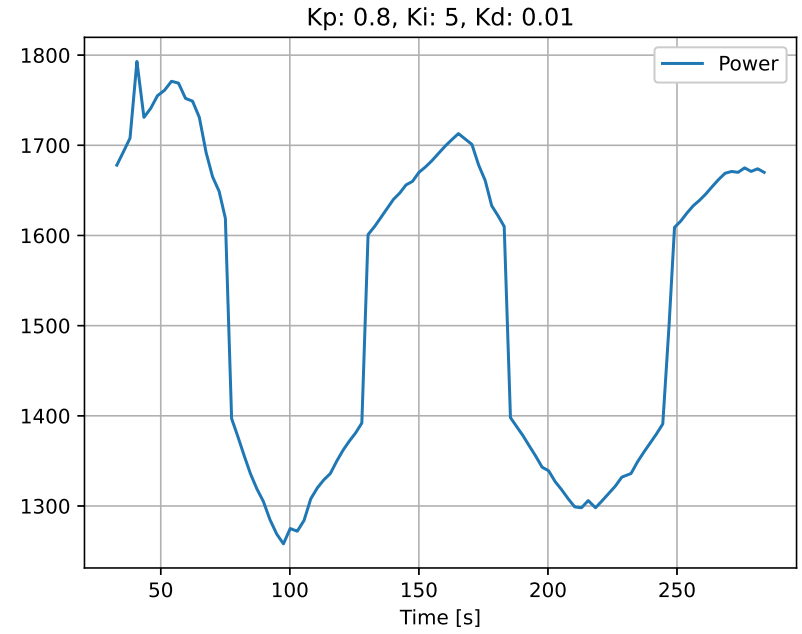
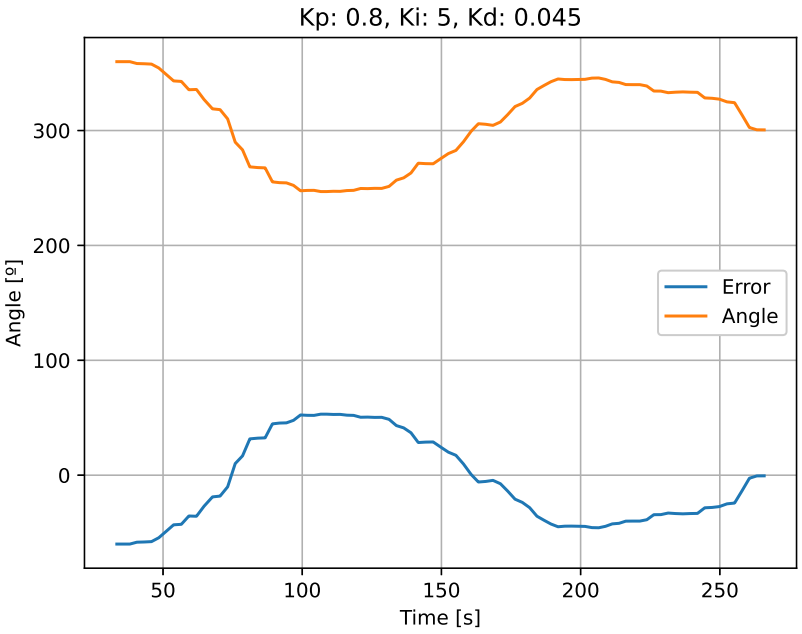


Figure B.11: Sensor data k_p: 0.8; k_i: 5; k_d: 1



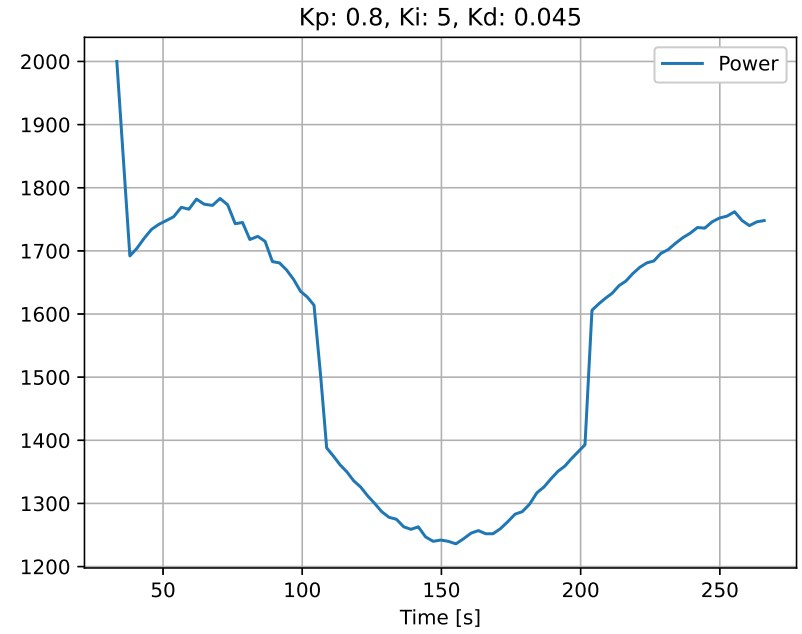


Figure B.12: Sensor data k_p: 0.8; k_i: 5; k_d: 1